Ege Ozan Özyedek
21703374
ozan.ozyedek@ug.bilkent.edu.tr

1 Question 1

1.a Preprocessing Data and Sample Images

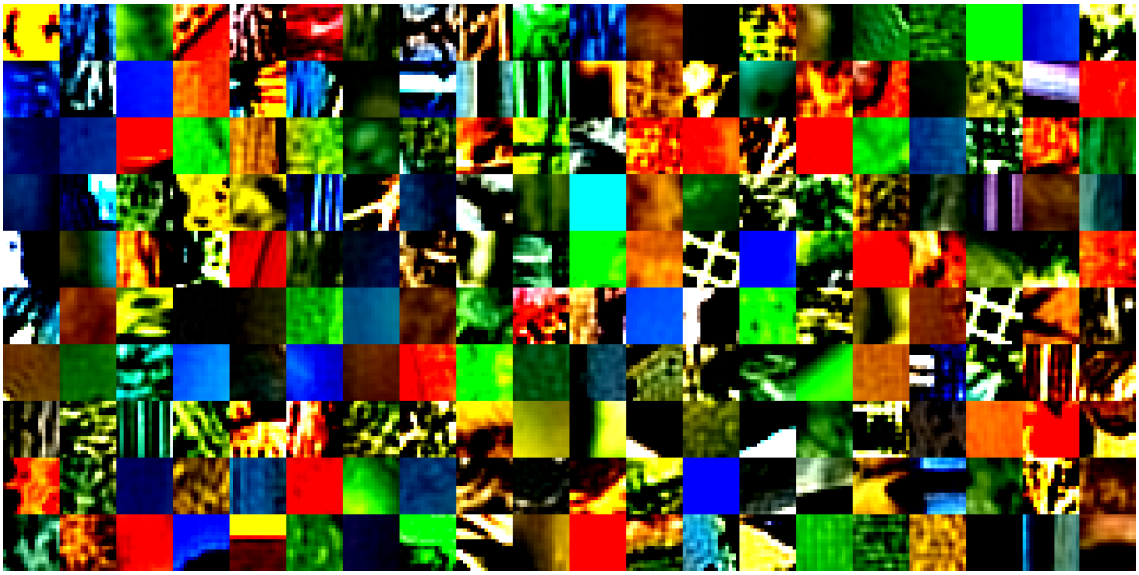


Figure 1: 200 randomly chosen RGB images from the given dataset.

Figure 2: 200 randomly chosen grayscaled images from the given dataset.

After doing the required steps to turn the images into grayscale I have obtained images similar to those displayed in Figure 2. The pictures in grayscale preserved the objects that were shown and only took their colors. Without color, the shapes of these objects are more apparent and the lines are much more defined. By defining these characteristics of the images we can now extract features.

1.b Discussion about the Code

Before talking about the features extracted, I will summarize the code written for this section

- A class named Q1AutoEncoder was written. This class has the required $aeCost$ function as well as the $solver$ function with the required inputs and outputs. The class initializes and
- The initialization is done by following the requirement given.

$$r = \sqrt{\frac{6}{L_{prev} + L_{next}}} \tag{1}$$

$$W = [-r, r],\ b = [-r, r] \tag{2}$$

- The weights $W1$ and $W2$ are the transposes of each other since this is an autoencoder and one of them is the encoder and the other is the decoder. They are initialized as the transpose of each other and at each update the code asserts that this holds true. The gradients of these are calculated as follows.

$$dW = \frac{dW1 + dW2^T}{2} \tag{3}$$

$$dW1 = dW \tag{4}$$

$$dW2 = dW^T \tag{5}$$

- Training is done with the sigmoid function. The loss function is applied as it is given in the assignment.
- The average activation term $\hat{\rho}_b$ found in KL is calculated by taking the average of h (the hidden layer activation) over all N training samples.

$$\hat{\rho}_b = \frac{1}{N} \sum_{n=0}^{N} h_n \tag{6}$$

- SGD with mini-batch is used. Using mini-batches gave better results.
- The optimized network parameters are found as
  - $\rho = 0.025$
  - $\beta = 2$
  - $L_{hid} = 64$
  - $\lambda = 5 * 10^-4$
  - $\eta = 0.075$
  - $\alpha = 0.85$
  - Batch Size = 32
  - Epoch = 200

1.c Hidden Layer Features

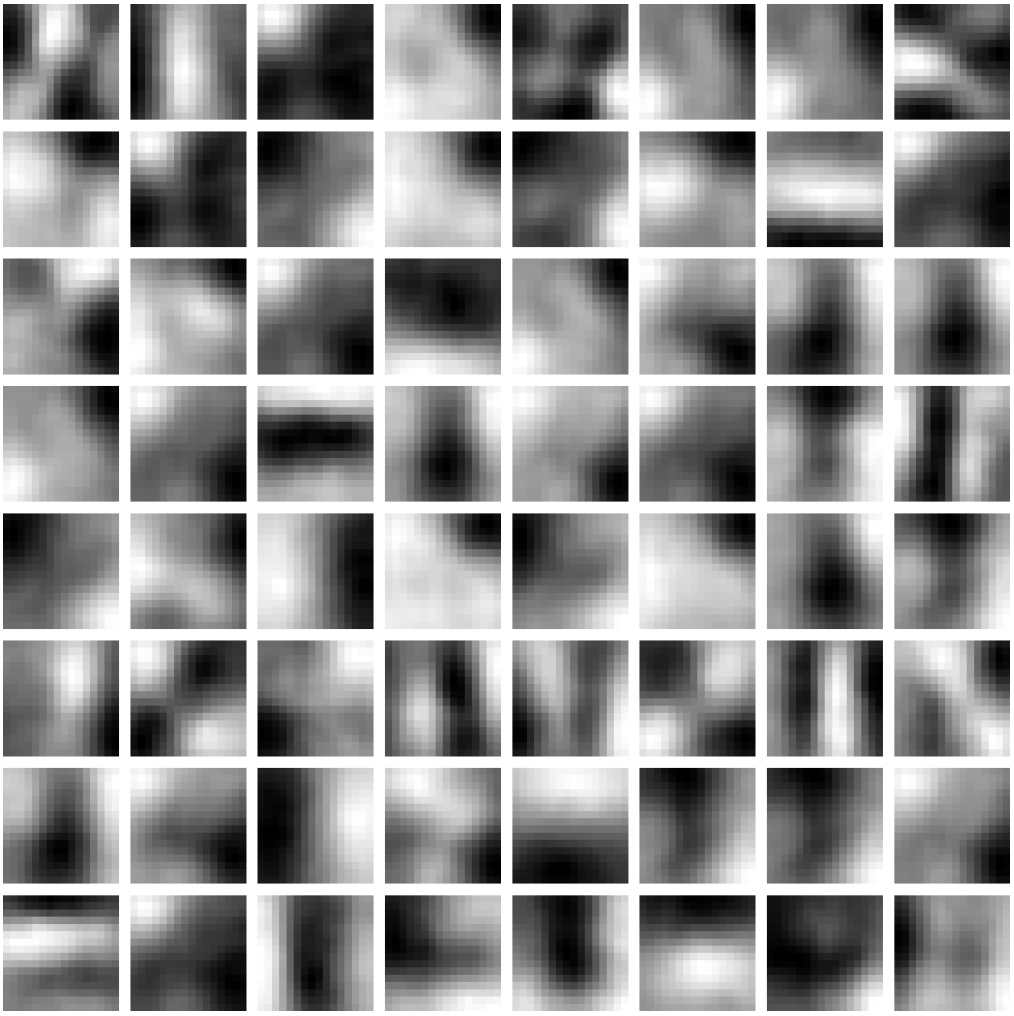


Figure 3: First layer weights plotted as images.

Above, the first weight matrix ($W1$) is displayed. The images resemble lines with depth in different orientations. There some that are curved lines as well, again with varying orientations. Although these pictures are not representative of natural images, meaning they do not visualize a natural image, they are however parts that create a natural image. In Figure 2 we can observe many rectangular shaped objects which have depth to them, the network can use these lines that it has picked up from training with the right inputs to reconstruct those rectangular images. Again, for the curved lines, there exists some images that have more circular features. The network can then recreate the pictures using these curved features.

1.d Different L_{hid} and λ Values

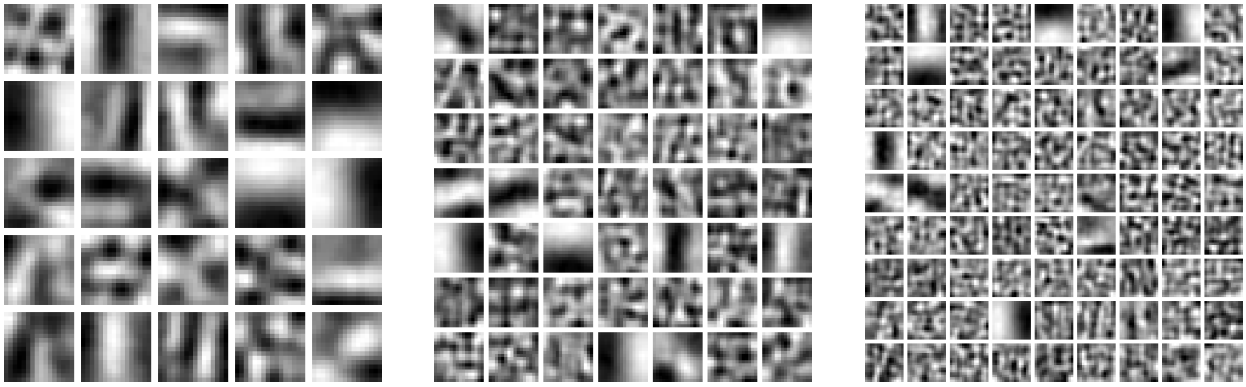
Below, 9 different combinations of images for 3 different L_{hid} and 3 different λ values can be found. The values are $L_{hid} = [25, 49, 81]$ and $\lambda = [10^{-3}, 10^{-4}, 0]$. The observations from these images are

- As the hidden unit size L_{hid} increases, different features are extracted. This is desirable, since we would want different representations of objects.
- The increase in L_{hid} might also find redundant features. As an example we can give $L_{hid} = 81$ in Figure 6, there exist similar diagonal features. Although this is good, if we increase the hidden size a bit more we might encounter more redundant features. This is obviously not desired because increasing feature dimension decreases performance.
- Increase in L_{hid} also means the data fits quicker compared to lower values, this is evident in Figure 4 and 5 with $L_{hid} = 81$, it overfits the data drastically and loses meaningful features. Hence higher hidden units overfit quickly.
- The regularization hyperparameter λ's change in value affects the outcome in various ways
  - $\lambda = 0$ causes overfitting since no regularization occurs. This is also true for lower values of the parameter.
  - $\lambda = 10^{-4}$ although gives better results compared to the zero case, the data is still overfit. Perhaps this shows the sensitivity of this parameter, since multiplying it by 10 will give us the next value which manages to extract meaningful features.
  - $\lambda = 10^{-3}$ seems to be the best parameter value, as for all hidden unit amounts it displays multiple quality features.

We can hence reduce these remarks to these two conclusions

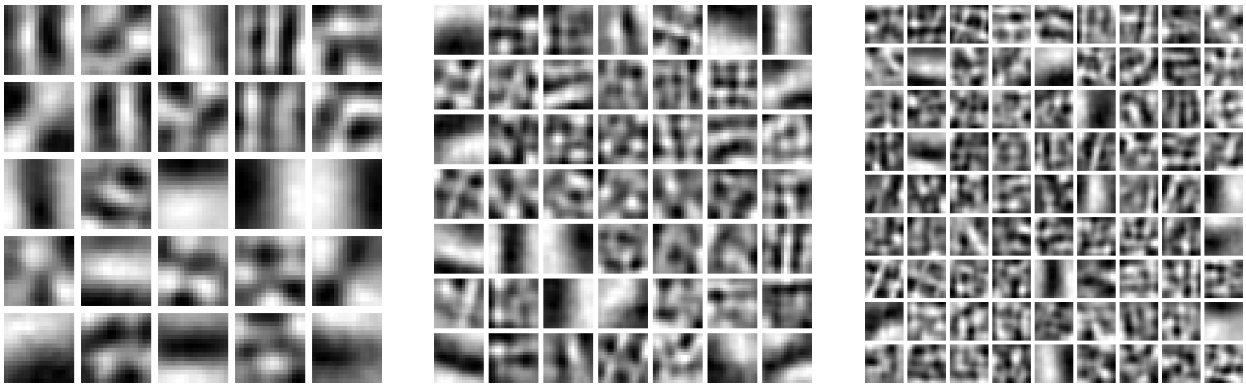
- The λ value should be chosen such that its not too big or too small, the two extremes which cause underfitting and overfitting respectively. This term seems sensitive so its selection should be done carefully. For our problem $\lambda = 10^{-3}$ seems to work better than $\lambda = 10^{-4}$; however bigger values such as $\lambda = 10^{-2}$ cause underfitting according to my tests.
- L_{hid} should be chosen as high as possible but the possibility of redundant features as well as overfitting should be kept in mind.

From these conclusions, I believe the best case out of the 9 extracted features is where $\lambda = 10^{-3}$ and $L_{hid} = 49$ or $L_{hid} = 81$, these features both represent multiple different edges and also vary in their orientation and representation.



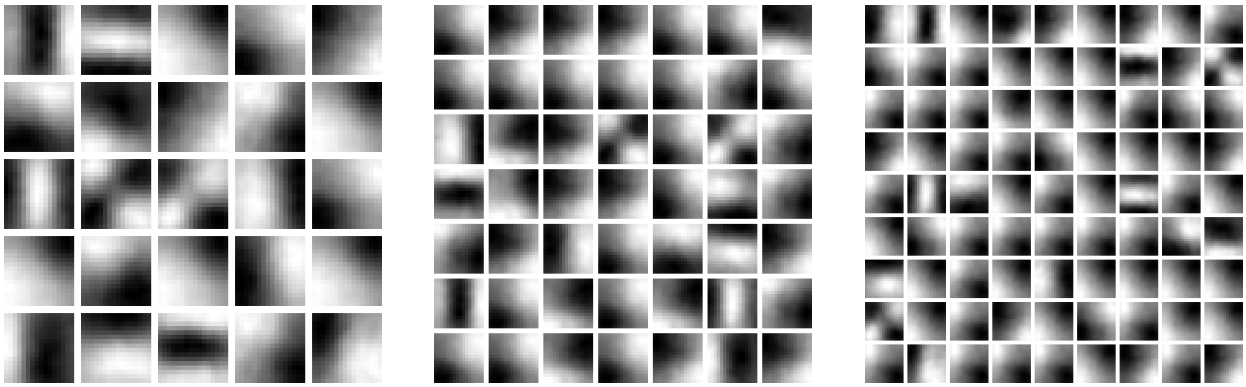
(a) $L_{hid} = 25$ (b) $L_{hid} = 49$ (c) $L_{hid} = 81$

Figure 4: Different hidden units for $\lambda = 0$



(a) $L_{hid} = 25$ (b) $L_{hid} = 49$ (c) $L_{hid} = 81$

Figure 5: Different hidden units for $\lambda = 10^{-4}$



(a) $L_{hid} = 25$ (b) $L_{hid} = 49$ (c) $L_{hid} = 81$

Figure 6: Different hidden units for $\lambda = 10^{-3}$

2 Question 2

2.a ConvolutionalNetworks

The notebook starts by importing the CIFAR-10 dataset which contain different classes of images. The convolutional neural network will be implemented on this dataset.

First, the naive convolution layer in which the convolution operation is implemented goes through forward pass. There are some parameters that should be noted: padding and strides. Padding as the name suggest covers the images outer perimeter with tthe given dimension of zeros, this way while convolving the possibility of losing pixels is reduced. This parameter is chosen as 1 for the forward pass. Stride is another parameter, and it changes the convolution windows traversal. For example, if stride is 2, then the window will move 2 blocks both for height and width. This has the effect of decreasing resolution of the images.

Then, to visualize the effect of the convolution two cute dog and cat images are used. Two image processing operations are done on these images, grayscaling and edge detection. To achieve this, two different filters are used, which is another type of parameter used in convolutions. These images are then displayed tto show what the convolution operation is capable of. After this the backward pass of the naive convolution layer is done, the output displays the relative error of the gradients.

After the convolutional layer, max pooling is done. The parameters pool width and height sets the pooling window. In the notebook, both are set to 2, hence the feature dimension width and height will halve. Pooling is used to summarize features, and by pooling the features become more generalized and will output similar results when changes in the input occur. Max pooling is a type of pooling, which calculates the maximum value of the pooling window. After the forward pass of the max pooling layer, the backward pass is done and the relative error is displayed.

After the simple implementations of convolution and pooling, the notebook provides ”Fast Layers”, which use the coding language C based Cython to speed up operations. The fast convolution and pooling layers then are evaluated and several evaluation metrics are displayed as the output. Then, ”sandwich” layers which combine convolution, the non-linearity (ReLU in the notebook) and pooling into one function. These sandwich layers output several evaluation metrics similarly.

A three layer network called Three-Layer ConvNet is defined. As maintenance several checks are done such as overfitting small dataset, gradient and loss sanity checks. After maintenance is done, the network is trained and the accuracy is found as around 50%. The first layer filters are visualized and displayed.

In the next section, the notebook gives some info on spacial batch normalization, which is a customized version of batch normalization that inputs and outputs arrays shaped (N, C, H, W) where N is the mini batch size. The spacial normalization is also presented as a network layer with forward and backwards passes. The forward pass is done and shapes, standard deviation and means of data is presented. The backward pass is also implemented and the relative error of the gradient is displayed. As the final part of the notebook group normalization is mentioned. It is presented as an alternative to layer normalization which does not work well with CNNs, as the cited article have found. Similar to spacial batch normalization, group normalization also has forward and backward pass functions and these are implemented and tested.

2.b PyTorch

This notebook serves the purpose of introducing the PyTorch library, which is a deep learning framework that can be used to create a variety of different models. As always, the notebook starts off by loading the CIFAR-10 image dataset.

PyTorch uses an array model called tensor which is very similar to a numpy array. Tensors can be used with CPUs and also GPUs, which increases the efficiency of matrix multiplications. The notebook starts by creating a $flatten$ function, which flattens tensors more than 2 dimensions into a 2 dimensional matrix. This is a practice that we already use in previous and this assignment also. This function is tested and results show that it works as intended. Then, a function called two_layer_fc is created with the goal of creating a 2 layered fully connected network with ReLU as the activation function. It uses PyTorch's funtional library to create the network. First the input is flattened and the weight parameters are initialized with the given parameters. As the notebook states, we don't need to store gradient values since PyTorch handles intermediate values. This is a sttrong advantage of libraries such as PyTorch and makes them more accessible and easy to use. The function returns the activation potential at the last layer. The function is then tested to assert that the output shape is as expected.

The notebook then moves onto the implementation of a three layer network. The function created for this model again takes input data and parameters as inputs. This model is a convolutional model, and contains "sandwiched" layers that we have seen in the CNN notebook. ReLU layers are used as activation functions. This function is again tested with a test function, and the results are as expected.

The notebook then displays two different initialization functions. One randomly initializes given weight and another initializes the input to zeros. Two other useful functions for finding the accuracy and for the training stage are also implemented. Using these functions, the two layer fully connected network and the three layer convolutional network are trained. The accuracy values reached are around the expected values.

Then the Model API of PyTorch is put on display. This trivializes the functions that were done beforehand in the notebook and uses PyTorch's own functions for layers. Two classes are coded for a two layer fully connected and three layer convolutional network, similar to the previous part of the notebook. Similar to the previous part, two functions for checking accuracy and training are coded. Then these two models are trained and tested with the accuracy value. The results are similar to the expected values.

For the next part, the Sequential API is introduced. This is the most convenient way of building a network with PyTorch and really trivializes the process. The Sequential model is created by stacking network layers and specifying an optimizer. So for example, if we were to code Q3 of this assignment we would add an LSTM layer, 3 MLP layers and set the optimizer to SGD. Again, we don't need to worry about gradients or updates as the model will handle these correctly. Using the Sequential API, the previously coded two layer fully connected and three layer convolutional networks are created and trained. The results are better than previous implementations.

The final part of the network has a challenge. A model is already coded, this model has three convolutional sequential models (which can be interpreted as convolutional "sandwich" layers) with batch normalization prior to the convolutional la ReLU as the activation function and max pooling at the end. Three of these sandwiched layers and a linear layer is then coupled together to create a PyTorch Sequential model. This model is trained on the database and it gives 73% test accuracy, which compared to previous results is a very successful.

I did some tests and managed to increase the test accuracy to 73.4%. To do this I did not touch the convolutional layers and simply extended the fully connected layers, named layer 4. I used a dropout layer after the convolutional layers to decrease the chance of overfitting with drop rate 0.3. This also allowed me to use a larger learning rate since I dropped the chance of overfitting significantly. After the dropout layer, I used a ReLU layer to increase complex learning. Its not too much of an increase but it was great practice that will prove beneficial in the final project.

3 Question 3

This question requires the implementation of a network which classifies human activity by using incoming data from three different sensors over T = 150 time samples. A simple rundown of the code, and some preliminary ideas on the MLP layers can be found below.

- The code architecture written for this question is called Q3NET: it creates the neural network and initializes the weights and biases according to the first layer choice and the preceding MLP layer sizes. Trains the network and has other functions that help in other operations such as predicting the output with the network weights.
- The code stops the training when the validation cross entropy at that run is in the range of ± 0.02 from the mean of the last 15 validation cross entropy values. For the report however, the whole 50 epoch run is displayed. This is more so for discussion and equal comparison. The code that forces the ending of the training run can be found at the end of the $train$ method in class Q3NET.
- For equal comparison, the hyperparameters were chosen as $\eta = 0.01, \alpha = 0.85$, Batch Size = 32, Hidden Layers = $[32, 16]$. This optimizes GRU, and also gives near optimal results for recurrent and LSTM layers.
- The initialization is done by following the Xavier Uniform distribution, which is as follows.

$$r = \sqrt{\frac{6}{L_{prev} + L_{next}}} \tag{7}$$

$$W = [-r, r],\ b = 0 \tag{8}$$

- For the MLP layers, the forward propagation is as follows

$$h = \phi(X \cdot W + b) \tag{9}$$

  where X is the input, W is the weight and b is the bias of the layer. The back propagation is as follows

$$\frac{dE}{dW} = h^T \cdot \delta \tag{10}$$

$$\frac{dE}{db} = [1]_{1xL} \cdot \delta \tag{11}$$

  where δ is the error gradient that is updated at each layer. For more information on how the error gradient is implemented, the written code is appended to the end of this report. The MLP is implemented the same way as previous assignments, but for completeness' it is included in this report.
- Hidden MLP layers use the ReLU function, the output layer uses a softmax sigmoid function paired with the cross entropy loss function.
- Three different first layers will be used, these are: Recurrent, LSTM and GRU layers. The problem (and the report) is sectioned such that each first layer is the focus of the discussion.
- The mathematical expressions of each first layer will be mentioned briefly in each section. For the backpropagation implementations the written code appended to the end of the report should be observed.
- Backpropagation Through Time (BPTT) is used for Recurrent, LSTM and GRU layers. It can be summarized with the following equation where W_{update} is the gradient weight update of the layer. To note, updates of biases or any multiplier in recurrent layers (LSTM and GRU included) are found the same way.

$$W_{update} = \sum_{t=t_0}^{T} \frac{dE(t)}{dW} \tag{12}$$

- The updates of weights are done by (for all layers, recurrent and MLP)

$$W = W - (\eta W_{update} + \alpha W_{momentum}) \tag{13}$$

  where $W_{momentum}$ is the momentum term that is updated at each batch.
- As a final note $h(t = 0)$ (RNN and GRU) and $c(t = 0)$ (LSTM) are taken to be equal to zero.

3.a Recurrent Layer

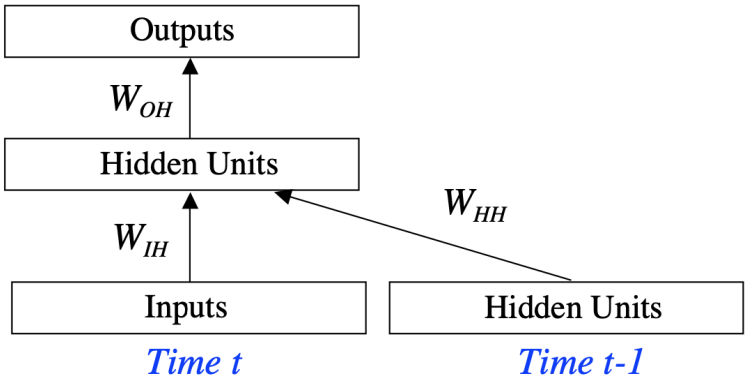


Figure 7: Schematic of a recurrent layer.

Reccurent neural networks can be imagined as an extension of a multi-layer perceptron, with a small change which is the fact that the RNN structure takes in both the input and the past output(s) of itself as an input. The mathematical interpretation of the RNN at each time step t for our problem is as follows

$$h(t) = tanh(\, x(t) \cdot W_{ih} + h(t-1) \cdot W_{hh} + b \,) \tag{14}$$

The error metrics plots as well as the confusion matrices can observed below. It can be seen that the results are not that great and far from desired. First thing that catches the eye is the way the loss and accuracy plots act unstable. There does not seem to be a direct progression towards a smaller loss and at times the loss increases. In other words, the network never converges to a stable value. Additionally, the confusion matrices show that the network only learns a portion of the data, hence cannot predict most of the classes correctly, both for train and test data, which might be an indication that the network is not learning. There might be several reasons as to why both of these problems occur, but one obvious reason is the problem of vanishing/exploding gradients.

The unstable nature of the loss comes from the fact that the gradients of network weights get very high or very low at a learning step (in my case the gradients were becoming zero), which causes the network to not learn the batch its training on if the gradients vanished, or outright crashing the program its running on by outputting NaN values if the gradient explodes. The gradient explodes most probably because of the accumulative nature of BPTT, and since our network has to go through 150 time samples, its understandable that it can accumulate to high values.

There are several possible fixes to this problem by changing the hyperparameters, one example might be to reduce the learning rate to make the network less susceptible to this problem. When the learning rate is slow the accumulation of the weight parameters slows down. However, there has to be an optimal point, since reducing the learning rate also means slowing down the learning process. If this parameter is reduced too much, the program might outright not learn any of the data its being trained on. So there still exists a chance of this problem arising.

How then, can this problem get eliminated/reduced such that we can fit the training data properly to the network and get stable results? The next two sections discuss the results of two architectural solutions to this problem, LSTM and GRU.

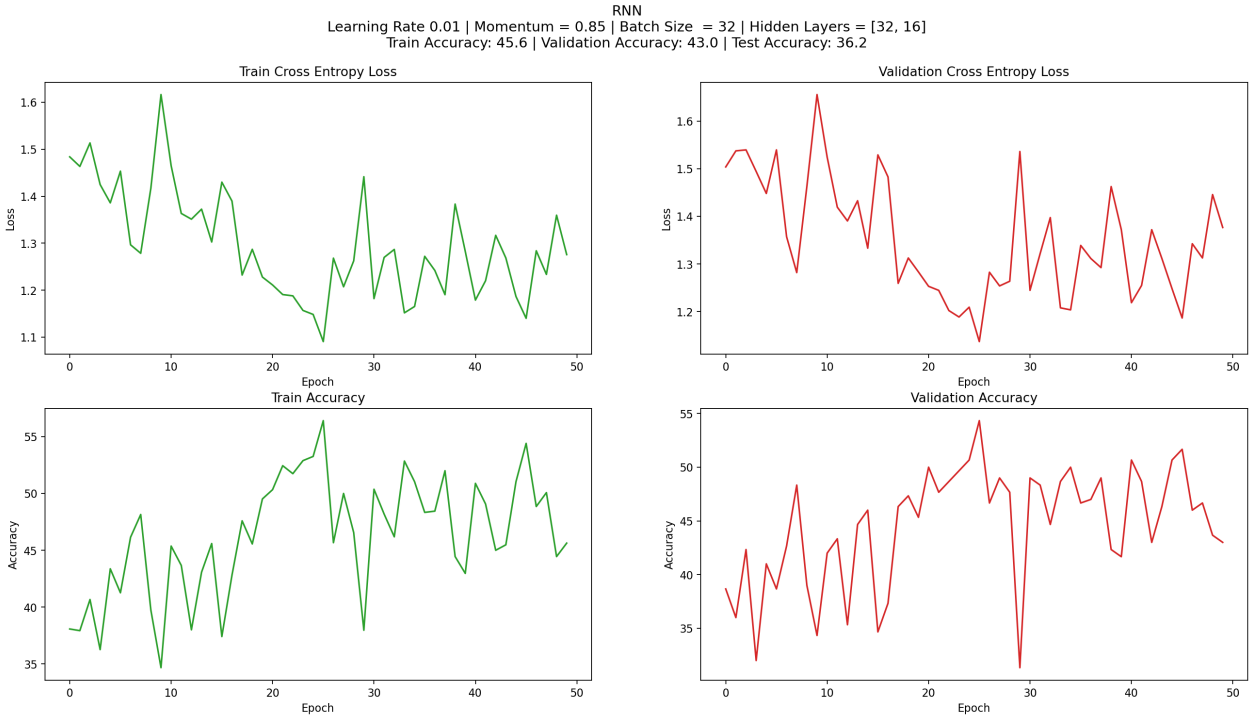


Figure 8: Cross Entropy and Accuracy Plots of Train and Validation Data for RNN. The hyperparameters, as well as the final accuracy values are displayed on top of the figure. Test Accuracy = 36.2%

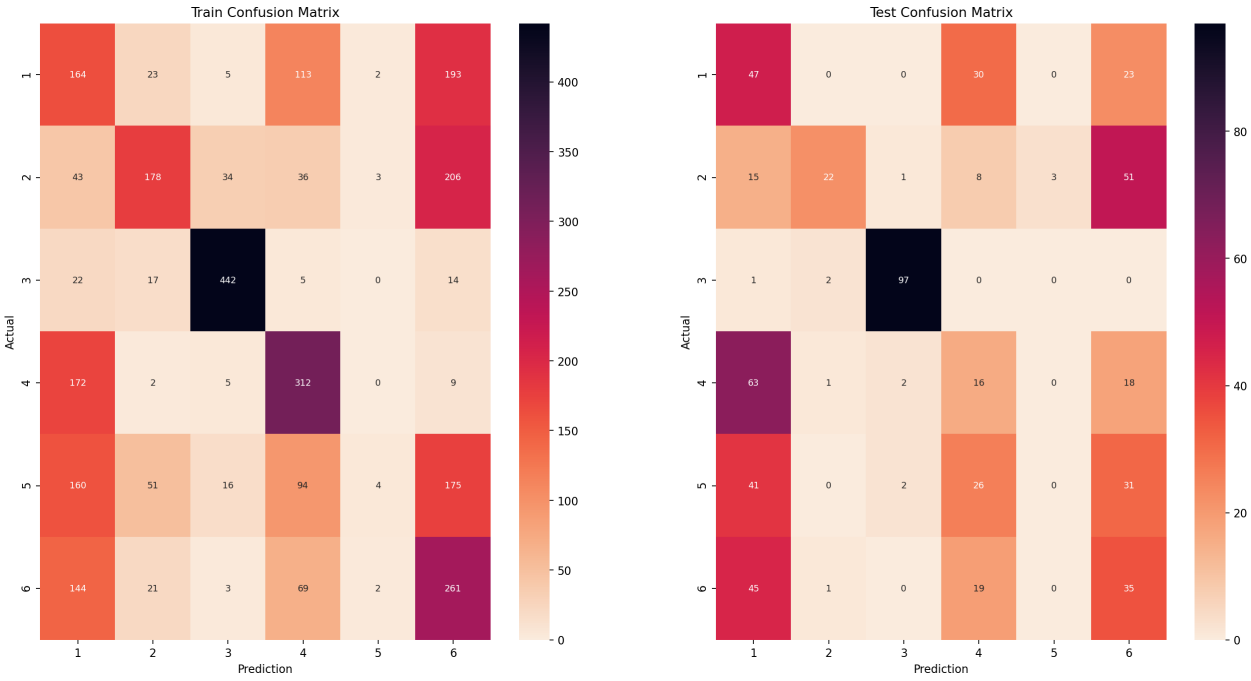


Figure 9: Confusion Matrices of Train and Test Data for RNN

3.b LSTM (Long-Short Term Memory) Layer

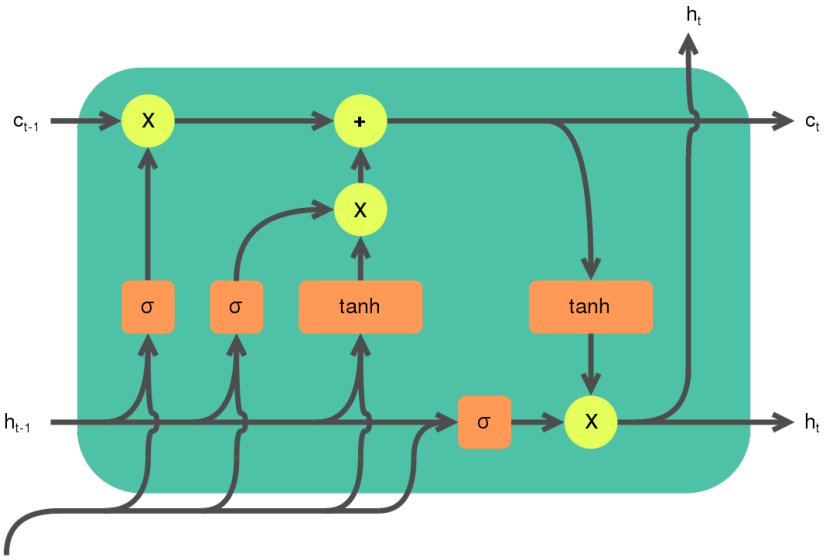


Figure 10: Schematic of a LSTM cell.

LSTM is created to solve (or reduce) the issues that arie from recurrent layers such as the vanishing/exploding gradient problem. Above the LSTM cell structure can be observed, below the mathematical interpretation of each cell, which are the forward propagation equations, can be viewed below.

$$hf(t) = \sigma(\ [h(t-1), x(t)] \cdot W_f + b_f\ ) \tag{15}$$

$$hi(t) = \sigma(\ [h(t-1), x(t)] \cdot W_i + b_i\ ) \tag{16}$$

$$hc(t) = tanh(\ [h(t-1), x(t)] \cdot W_c + b_c\ ) \tag{17}$$

$$ho(t) = \sigma(\ [h(t-1), x(t)] \cdot W_o + b_o\ ) \tag{18}$$

$$c(t) = hf(t) * c(t-1) + hi(t) * hc(t) \tag{19}$$

$$h(t) = ho(t) * tanh(c(t)) \tag{20}$$

The above equations represent several gates inside of the LSTM structure that selectively forgets and saves past values of itself. This way the past value of the network is much more stable and the updates also reflect this.

Looking at the train and validation loss plots we can see a huge improvement over the recurrent layer. This improvement both reflects the stability of the loss progression, and also the values of accuracy as well. The test accuracy is 70.3% in this model, which is a huge increase from recurrent layers 36.2%. The confusion matrices also show that the network learns the class patterns and correctly does predictions.

It is also apparent that the problem of vanishing gradients is eliminated. Although the learning rate is selected to be 0.01 for the graphs higher learning rates can potentially give better results, without the effect of vanishing/exploding gradients. For example I tried several different learning rates, and the best was with $\eta = 0.1$, which gave around 79% for the test accuracy. Comparatively, its nearly impossible to use such a high learning rate for the recurrent layer as it will become much more unstable than it already is, or perhaps give an overflow error by outputting exploding gradients. Hence we can conclude that LSTM's eliminate the problem of vanishing gradients, and is much less susceptible to changes in the gradient updates.

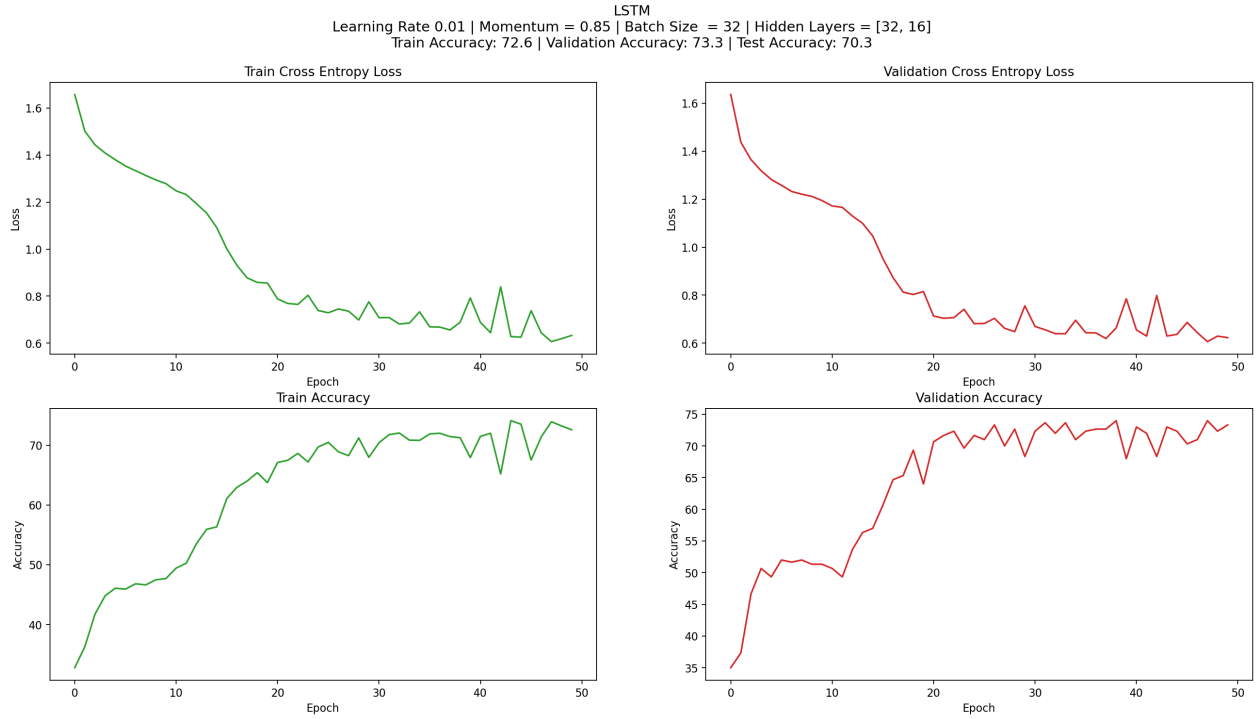


Figure 11: Cross Entropy and Accuracy Plots of Train and Validation Data for LSTM. The hyperparameters, as well as the final accuracy values are displayed on top of the figure. Test Accuracy = 70.3%

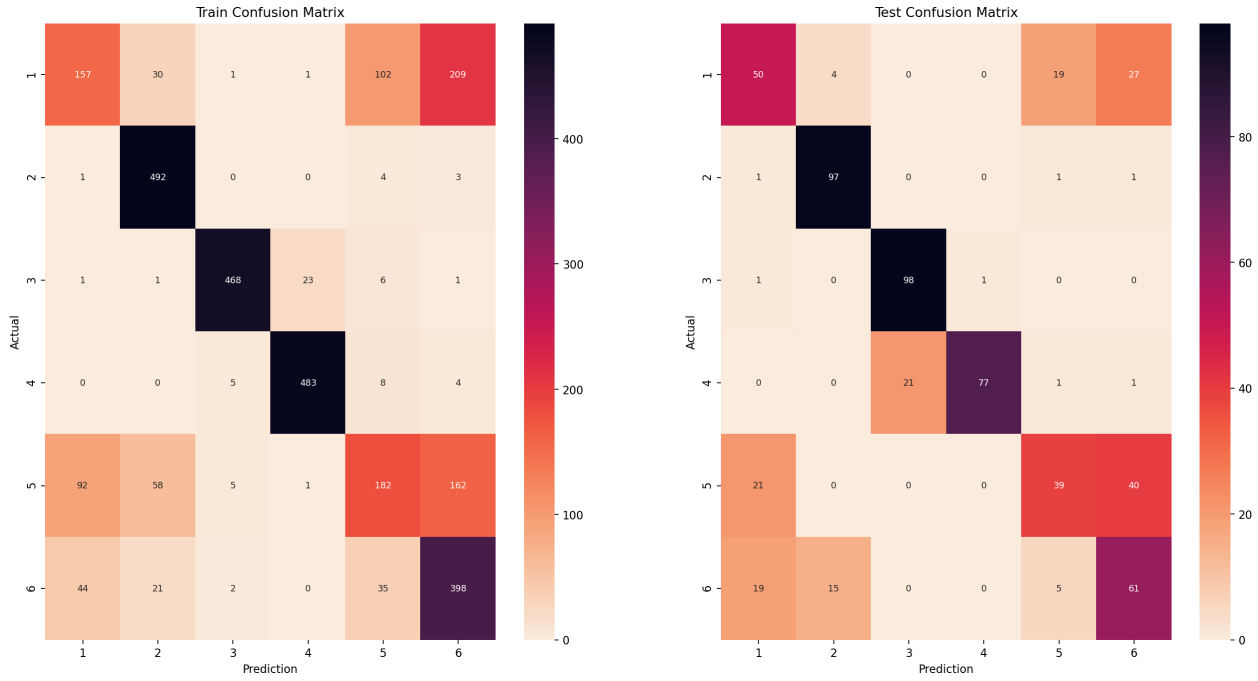


Figure 12: Confusion Matrices of Train and Test Data for LSTM

3.c GRU (Gated Recurrent Unit) Layer

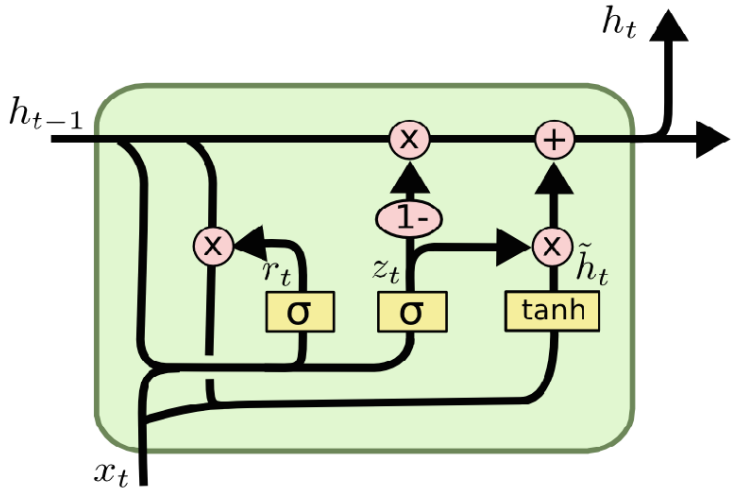


Figure 13: Schematic of a GRU cell.

GRU is another cell-based structure similar to LSTM that includes less gates (and equations) than LSTM, while still giving exceptional performance. Below, the mathematical expressions for this layer can be found.

$$z(t) = \sigma(\, x(t) \cdot W_z + h(t-1) \cdot U_z + b_z \,) \tag{21}$$

$$r(t) = \sigma(\, x(t) \cdot W_r + h(t-1) \cdot U_r + b_r \,) \tag{22}$$

$$\tilde{h}(t) = \tanh(\, x(t) \cdot W_h + (h(t-1) * r(t)) \cdot U_h + b_h \,) \tag{23}$$

$$h(t) = (1 - z(t)) * h(t-1) + z(t) * \tilde{h}(t) \tag{24}$$

The results show that GRU is the best performer both in terms of classification and loss/accuracy results. Out of the three RNN layers that were implemented., GRU has the highest test accuracy at 81%. Observing the loss plots it can be observed that its again very stable compared to the recurrent layer and the network does learn correct predictions, hence its safe to say that GRU also reduces the problem of vanishing gradients.

The comparison between LSTM and GRU is an ongoing research topic, and they differ in performance for various reasons. Naturally, both have their advantages and disadvantages and one might be better suited for a specific problem while the other is not as convenient.

- LSTM uses three gates (input: hi, output:ho, forget:hf) while GRU uses two (update: z, reset: r). The gate difference has many implications and one of the main reasons for the difference in performance
- GRU is much faster to train compared to LSTM. For 50 epochs and a batch size of 32, LSTM takes around 22 minutes while GRU takes about 17 minutes. Since it has less parameters, it doesn't waste a lot of time multiplying matrices compared to LSTM. We can also see the effect of this by looking at the test accuracy, in the same 50 epoch run GRU learns the data faster and reaches 81% accuracy.
- GRU is much more memory efficient since it doesn't require the additional gate that LSTM includes.
- LSTM is much more stable and less susceptible to the effects of vanishing/exploding gradients. I experienced this first hand as GRU cannot work with high learning rates ($\eta = 0.1$ gives overflow for GRU, at least for this problem) because it does not eliminate vanishing/exploding gradient entirely, only reduces the effects of it. This is most evident on long time samples similar to/bigger than the dataset given to us, which has 150 time samples.

Observing the above evaluations, we can conclude that;

- GRU is more preferable if the data has less time samples and the program is desired to be more efficient.
- LSTM is more preferable if the data has more time samples and the program is desired to be accurate.

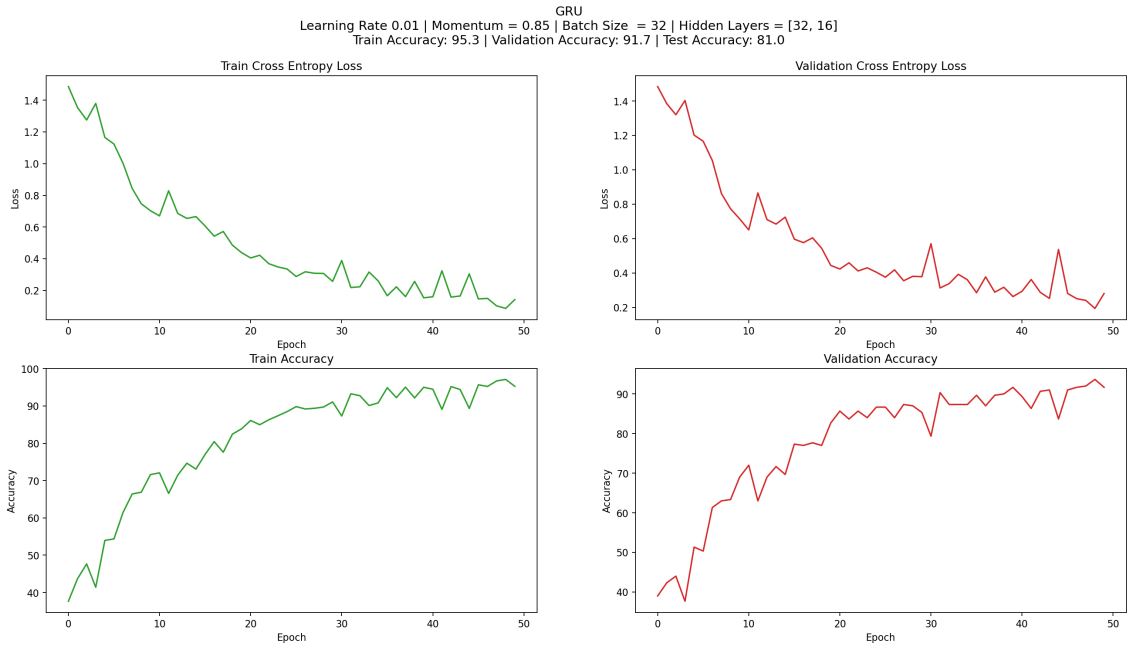


Figure 14: Cross Entropy and Accuracy Plots of Train and Validation Data for GRU. The hyperparameters, as well as the final accuracy values are displayed on top of the figure. Test Accuracy = 81.0%

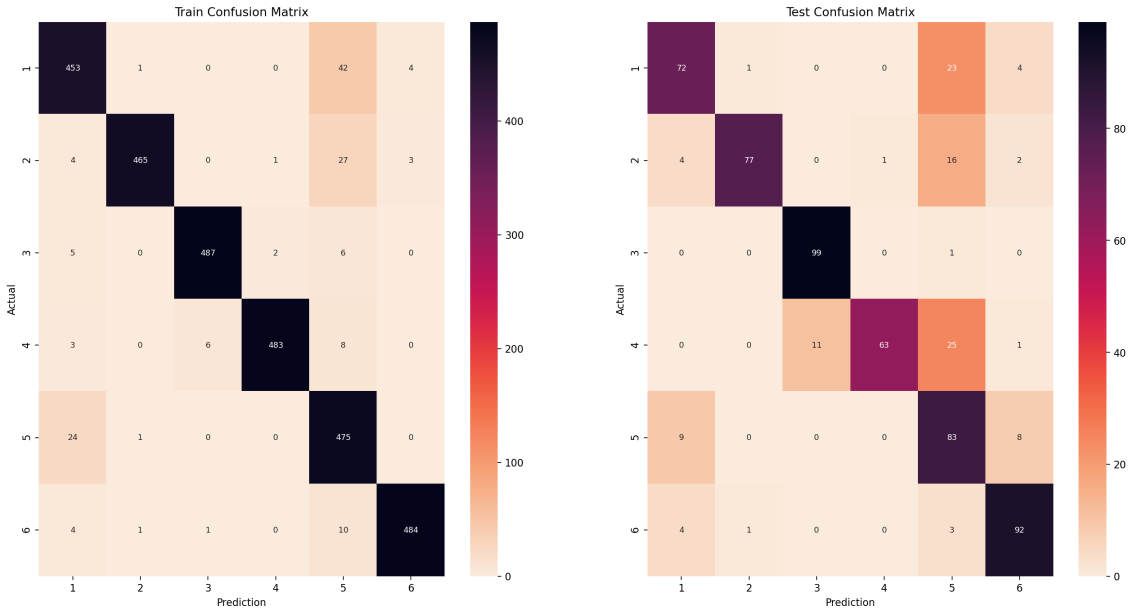


Figure 15: Confusion Matrices of Train and Test Data for GRU

4 Output of ConvolutionalNetworks.ipynb

The following pages in this section contain the output of the file ”ConvolutionalNetworks.ipynb.

1 Convolutional Networks

So far we have worked with deep fully-connected networks, using them to explore different optimization strategies and network architectures. Fully-connected networks are a good testbed for experimentation because they are very computationally efficient, but in practice all state-of-the-art results use convolutional networks instead.

First you will implement several layer types that are used in convolutional networks. You will then use these layers to train a convolutional network on the CIFAR-10 dataset.

[4]:
```
# As usual, a bit of setup
import numpy as np
import matplotlib.pyplot as plt
from cs231n.classifiers.cnn import *
from cs231n.data_utils import get_CIFAR10_data
from cs231n.gradient_check import eval_numerical_gradient_array,␣
 ↪eval_numerical_gradient
from cs231n.layers import *
from cs231n.fast_layers import *
from cs231n.solver import Solver

%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

# for auto-reloading external modules
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2

def rel_error(x, y):
  """ returns relative error """
  return np.max(np.abs(x - y) / (np.maximum(1e-8, np.abs(x) + np.abs(y))))
```

[5]:
```
# Load the (preprocessed) CIFAR10 data.

data = get_CIFAR10_data()
for k, v in data.items():
  print('%s: ' % k, v.shape)
```

```
X_train:  (49000, 3, 32, 32)
y_train:  (49000,)
X_val:  (1000, 3, 32, 32)
y_val:  (1000,)
X_test:  (1000, 3, 32, 32)
y_test:  (1000,)
```

2 Convolution: Naive forward pass

The core of a convolutional network is the convolution operation. In the file `cs231n/layers.py`, implement the forward pass for the convolution layer in the function `conv_forward_naive`.

You don't have to worry too much about efficiency at this point; just write the code in whatever way you find most clear.

You can test your implementation by running the following:

[6]:
```
x_shape = (2, 3, 4, 4)
w_shape = (3, 3, 4, 4)
x = np.linspace(-0.1, 0.5, num=np.prod(x_shape)).reshape(x_shape)
w = np.linspace(-0.2, 0.3, num=np.prod(w_shape)).reshape(w_shape)
b = np.linspace(-0.1, 0.2, num=3)

conv_param = {'stride': 2, 'pad': 1}
out, _ = conv_forward_naive(x, w, b, conv_param)
correct_out = np.array([[[[-0.08759809, -0.10987781],
                          [-0.18387192, -0.2109216 ]],
                         [[ 0.21027089,  0.21661097],
                          [ 0.22847626,  0.23004637]],
                         [[ 0.50813986,  0.54309974],
                          [ 0.64082444,  0.67101435]]],
                        [[[-0.98053589, -1.03143541],
                          [-1.19128892, -1.24695841]],
                         [[ 0.69108355,  0.66880383],
                          [ 0.59480972,  0.56776003]],
                         [[ 2.36270298,  2.36904306],
                          [ 2.38090835,  2.38247847]]]])

# Compare your output to ours; difference should be around e-8
print('Testing conv_forward_naive')
print('difference: ', rel_error(out, correct_out))
```

```
Testing conv_forward_naive
difference:  2.2121476417505994e-08
```

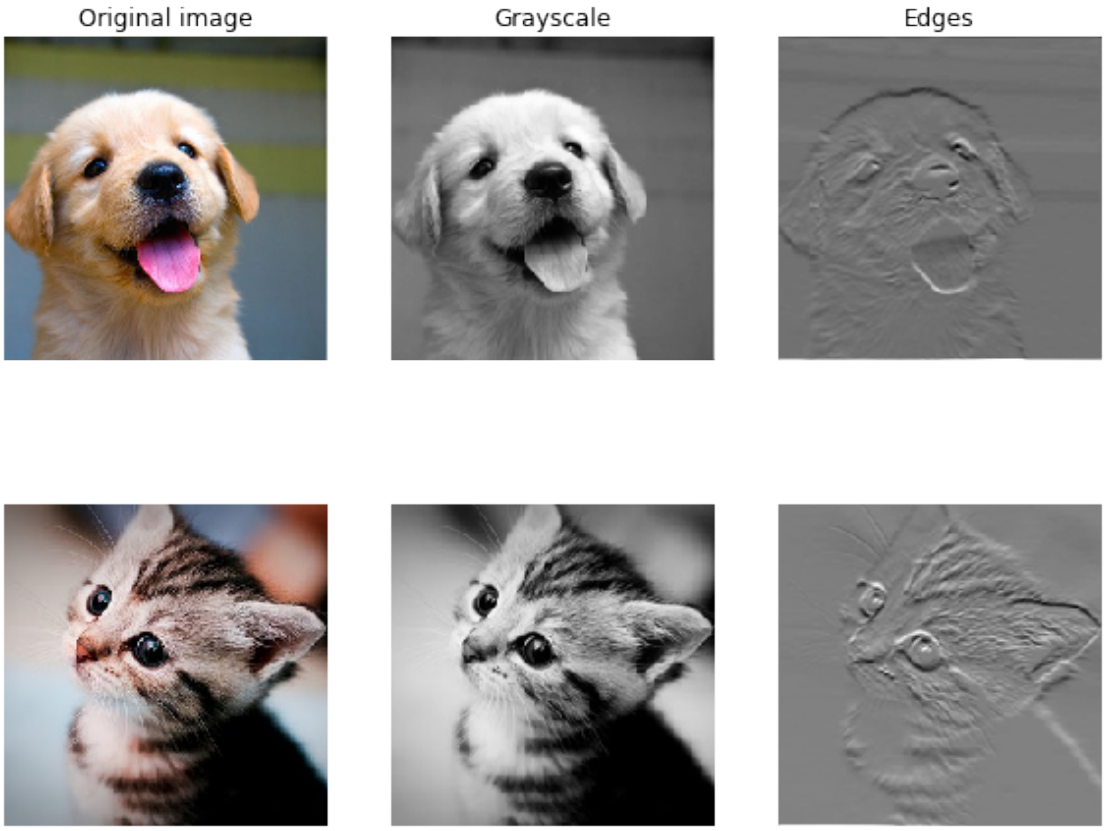
3 Aside: Image processing via convolutions

As fun way to both check your implementation and gain a better understanding of the type of operation that convolutional layers can perform, we will set up an input containing two images and manually set up filters that perform common image processing operations (grayscale conversion and edge detection). The convolution forward pass will apply these operations to each of the input images. We can then visualize the results as a sanity check.

[7]:
```
from imageio import imread
from PIL import Image
```

```
kitten, puppy = imread('kitten.jpg'), imread('puppy.jpg')
# kitten is wide, and puppy is already square
d = kitten.shape[1] - kitten.shape[0]
kitten_cropped = kitten[:, d//2:-d//2, :]

img_size = 200   # Make this smaller if it runs too slow
x = np.zeros((2, 3, img_size, img_size))
x[0, :, :, :] = np.array(Image.fromarray(puppy).resize((img_size, img_size))).
 ↪transpose((2, 0, 1))
x[1, :, :, :] = np.array(Image.fromarray(kitten_cropped).resize((img_size,␣
 ↪img_size))).transpose((2, 0, 1))

# Set up a convolutional weights holding 2 filters, each 3x3
w = np.zeros((2, 3, 3, 3))

# The first filter converts the image to grayscale.
# Set up the red, green, and blue channels of the filter.
w[0, 0, :, :] = [[0, 0, 0], [0, 0.3, 0], [0, 0, 0]]
w[0, 1, :, :] = [[0, 0, 0], [0, 0.6, 0], [0, 0, 0]]
w[0, 2, :, :] = [[0, 0, 0], [0, 0.1, 0], [0, 0, 0]]

# Second filter detects horizontal edges in the blue channel.
w[1, 2, :, :] = [[1, 2, 1], [0, 0, 0], [-1, -2, -1]]

# Vector of biases. We don't need any bias for the grayscale
# filter, but for the edge detection filter we want to add 128
# to each output so that nothing is negative.
b = np.array([0, 128])

# Compute the result of convolving each input in x with each filter in w,
# offsetting by b, and storing the results in out.
out, _ = conv_forward_naive(x, w, b, {'stride': 1, 'pad': 1})

def imshow_noax(img, normalize=True):
    """ Tiny helper to show images as uint8 and remove axis labels """
    if normalize:
        img_max, img_min = np.max(img), np.min(img)
        img = 255.0 * (img - img_min) / (img_max - img_min)
    plt.imshow(img.astype('uint8'))
    plt.gca().axis('off')

# Show the original images and the results of the conv operation
plt.subplot(2, 3, 1)
imshow_noax(puppy, normalize=False)
plt.title('Original image')
plt.subplot(2, 3, 2)
```

```
imshow_noax(out[0, 0])
plt.title('Grayscale')
plt.subplot(2, 3, 3)
imshow_noax(out[0, 1])
plt.title('Edges')
plt.subplot(2, 3, 4)
imshow_noax(kitten_cropped, normalize=False)
plt.subplot(2, 3, 5)
imshow_noax(out[1, 0])
plt.subplot(2, 3, 6)
imshow_noax(out[1, 1])
plt.show()
```



4 Convolution: Naive backward pass

Implement the backward pass for the convolution operation in the function `conv_backward_naive` in the file `cs231n/layers.py`. Again, you don't need to worry too much about computational efficiency.

When you are done, run the following to check your backward pass with a numeric gradient

check.

[8]:
```
np.random.seed(231)
x = np.random.randn(4, 3, 5, 5)
w = np.random.randn(2, 3, 3, 3)
b = np.random.randn(2,)
dout = np.random.randn(4, 2, 5, 5)
conv_param = {'stride': 1, 'pad': 1}

dx_num = eval_numerical_gradient_array(lambda x: conv_forward_naive(x, w, b,␣
 ↪conv_param)[0], x, dout)
dw_num = eval_numerical_gradient_array(lambda w: conv_forward_naive(x, w, b,␣
 ↪conv_param)[0], w, dout)
db_num = eval_numerical_gradient_array(lambda b: conv_forward_naive(x, w, b,␣
 ↪conv_param)[0], b, dout)

out, cache = conv_forward_naive(x, w, b, conv_param)
dx, dw, db = conv_backward_naive(dout, cache)

# Your errors should be around e-8 or less.
print('Testing conv_backward_naive function')
print('dx error: ', rel_error(dx, dx_num))
print('dw error: ', rel_error(dw, dw_num))
print('db error: ', rel_error(db, db_num))
```

```
Testing conv_backward_naive function
dx error:  1.159803161159293e-08
dw error:  2.2471264748452487e-10
db error:  3.37264006649648e-11
```

5 Max-Pooling: Naive forward

Implement the forward pass for the max-pooling operation in the function `max_pool_forward_naive` in the file `cs231n/layers.py`. Again, don't worry too much about computational efficiency.

Check your implementation by running the following:

[9]:
```
x_shape = (2, 3, 4, 4)
x = np.linspace(-0.3, 0.4, num=np.prod(x_shape)).reshape(x_shape)
pool_param = {'pool_width': 2, 'pool_height': 2, 'stride': 2}

out, _ = max_pool_forward_naive(x, pool_param)

correct_out = np.array([[[[-0.26315789, -0.24842105],
                          [-0.20421053, -0.18947368]],
                         [[-0.14526316, -0.13052632],
                          [-0.08631579, -0.07157895]],
```

```
                        [[-0.02736842, -0.01263158],
                         [ 0.03157895,  0.04631579]]],
                       [[[ 0.09052632,  0.10526316],
                         [ 0.14947368,  0.16421053]],
                        [[ 0.20842105,  0.22315789],
                         [ 0.26736842,  0.28210526]],
                        [[ 0.32631579,  0.34105263],
                         [ 0.38526316,  0.4       ]]]])

# Compare your output with ours. Difference should be on the order of e-8.
print('Testing max_pool_forward_naive function:')
print('difference: ', rel_error(out, correct_out))
```

```
Testing max_pool_forward_naive function:
difference:  4.1666665157267834e-08
```

6 Max-Pooling: Naive backward

Implement the backward pass for the max-pooling operation in the function `max_pool_backward_naive` in the file `cs231n/layers.py`. You don't need to worry about computational efficiency.

Check your implementation with numeric gradient checking by running the following:

[10]:
```
np.random.seed(231)
x = np.random.randn(3, 2, 8, 8)
dout = np.random.randn(3, 2, 4, 4)
pool_param = {'pool_height': 2, 'pool_width': 2, 'stride': 2}

dx_num = eval_numerical_gradient_array(lambda x: max_pool_forward_naive(x,␣
 ↪pool_param)[0], x, dout)

out, cache = max_pool_forward_naive(x, pool_param)
dx = max_pool_backward_naive(dout, cache)

# Your error should be on the order of e-12
print('Testing max_pool_backward_naive function:')
print('dx error: ', rel_error(dx, dx_num))
```

```
Testing max_pool_backward_naive function:
dx error:  3.27562514223145e-12
```

7 Fast layers

Making convolution and pooling layers fast can be challenging. To spare you the pain, we've provided fast implementations of the forward and backward passes for convolution and pooling layers in the file `cs231n/fast_layers.py`.

The fast convolution implementation depends on a Cython extension; to compile it you need to run the following from the `cs231n` directory:

```
python setup.py build_ext --inplace
```

The API for the fast versions of the convolution and pooling layers is exactly the same as the naive versions that you implemented above: the forward pass receives data, weights, and parameters and produces outputs and a cache object; the backward pass recieves upstream derivatives and the cache object and produces gradients with respect to the data and weights.

NOTE: The fast implementation for pooling will only perform optimally if the pooling regions are non-overlapping and tile the input. If these conditions are not met then the fast pooling implementation will not be much faster than the naive implementation.

You can compare the performance of the naive and fast versions of these layers by running the following:

[11]:
```
# Rel errors should be around e-9 or less
from cs231n.fast_layers import conv_forward_fast, conv_backward_fast
from time import time
np.random.seed(231)
x = np.random.randn(100, 3, 31, 31)
w = np.random.randn(25, 3, 3, 3)
b = np.random.randn(25,)
dout = np.random.randn(100, 25, 16, 16)
conv_param = {'stride': 2, 'pad': 1}

t0 = time()
out_naive, cache_naive = conv_forward_naive(x, w, b, conv_param)
t1 = time()
out_fast, cache_fast = conv_forward_fast(x, w, b, conv_param)
t2 = time()

print('Testing conv_forward_fast:')
print('Naive: %fs' % (t1 - t0))
print('Fast: %fs' % (t2 - t1))
print('Speedup: %fx' % ((t1 - t0) / (t2 - t1)))
print('Difference: ', rel_error(out_naive, out_fast))

t0 = time()
dx_naive, dw_naive, db_naive = conv_backward_naive(dout, cache_naive)
t1 = time()
dx_fast, dw_fast, db_fast = conv_backward_fast(dout, cache_fast)
t2 = time()

print('\nTesting conv_backward_fast:')
print('Naive: %fs' % (t1 - t0))
print('Fast: %fs' % (t2 - t1))
print('Speedup: %fx' % ((t1 - t0) / (t2 - t1)))
```

```
print('dx difference: ', rel_error(dx_naive, dx_fast))
print('dw difference: ', rel_error(dw_naive, dw_fast))
print('db difference: ', rel_error(db_naive, db_fast))
```

```
Testing conv_forward_fast:
Naive: 4.618436s
Fast: 0.008466s
Speedup: 545.527190x
Difference:  4.926407851494105e-11

Testing conv_backward_fast:
Naive: 6.063134s
Fast: 0.008765s
Speedup: 691.726308x
dx difference:  1.949764775345631e-11
dw difference:  3.9697679477805934e-13
db difference:  0.0
```

[12]:
```
# Relative errors should be close to 0.0
from cs231n.fast_layers import max_pool_forward_fast, max_pool_backward_fast
np.random.seed(231)
x = np.random.randn(100, 3, 32, 32)
dout = np.random.randn(100, 3, 16, 16)
pool_param = {'pool_height': 2, 'pool_width': 2, 'stride': 2}

t0 = time()
out_naive, cache_naive = max_pool_forward_naive(x, pool_param)
t1 = time()
out_fast, cache_fast = max_pool_forward_fast(x, pool_param)
t2 = time()

print('Testing pool_forward_fast:')
print('Naive: %fs' % (t1 - t0))
print('fast: %fs' % (t2 - t1))
print('speedup: %fx' % ((t1 - t0) / (t2 - t1)))
print('difference: ', rel_error(out_naive, out_fast))

t0 = time()
dx_naive = max_pool_backward_naive(dout, cache_naive)
t1 = time()
dx_fast = max_pool_backward_fast(dout, cache_fast)
t2 = time()

print('\nTesting pool_backward_fast:')
print('Naive: %fs' % (t1 - t0))
print('fast: %fs' % (t2 - t1))
print('speedup: %fx' % ((t1 - t0) / (t2 - t1)))
```

```
print('dx difference: ', rel_error(dx_naive, dx_fast))
```

```
Testing pool_forward_fast:
Naive: 0.155931s
fast: 0.002616s
speedup: 59.602752x
difference:  0.0

Testing pool_backward_fast:
Naive: 0.463956s
fast: 0.009750s
speedup: 47.585758x
dx difference:  0.0
```

8 Convolutional "sandwich" layers

Previously we introduced the concept of "sandwich" layers that combine multiple operations into commonly used patterns. In the file `cs231n/layer_utils.py` you will find sandwich layers that implement a few commonly used patterns for convolutional networks.

[13]:
```
from cs231n.layer_utils import conv_relu_pool_forward, conv_relu_pool_backward
np.random.seed(231)
x = np.random.randn(2, 3, 16, 16)
w = np.random.randn(3, 3, 3, 3)
b = np.random.randn(3,)
dout = np.random.randn(2, 3, 8, 8)
conv_param = {'stride': 1, 'pad': 1}
pool_param = {'pool_height': 2, 'pool_width': 2, 'stride': 2}

out, cache = conv_relu_pool_forward(x, w, b, conv_param, pool_param)
dx, dw, db = conv_relu_pool_backward(dout, cache)

dx_num = eval_numerical_gradient_array(lambda x: conv_relu_pool_forward(x, w, b,␣
 ↪conv_param, pool_param)[0], x, dout)
dw_num = eval_numerical_gradient_array(lambda w: conv_relu_pool_forward(x, w, b,␣
 ↪conv_param, pool_param)[0], w, dout)
db_num = eval_numerical_gradient_array(lambda b: conv_relu_pool_forward(x, w, b,␣
 ↪conv_param, pool_param)[0], b, dout)

# Relative errors should be around e-8 or less
print('Testing conv_relu_pool')
print('dx error: ', rel_error(dx_num, dx))
print('dw error: ', rel_error(dw_num, dw))
print('db error: ', rel_error(db_num, db))
```

```
Testing conv_relu_pool
dx error:  6.514336569263308e-09
```

```
dw error:  1.490843753539445e-08
db error:  2.037390356217257e-09
```

[14]:
```
from cs231n.layer_utils import conv_relu_forward, conv_relu_backward
np.random.seed(231)
x = np.random.randn(2, 3, 8, 8)
w = np.random.randn(3, 3, 3, 3)
b = np.random.randn(3,)
dout = np.random.randn(2, 3, 8, 8)
conv_param = {'stride': 1, 'pad': 1}

out, cache = conv_relu_forward(x, w, b, conv_param)
dx, dw, db = conv_relu_backward(dout, cache)

dx_num = eval_numerical_gradient_array(lambda x: conv_relu_forward(x, w, b,␣
 ↪conv_param)[0], x, dout)
dw_num = eval_numerical_gradient_array(lambda w: conv_relu_forward(x, w, b,␣
 ↪conv_param)[0], w, dout)
db_num = eval_numerical_gradient_array(lambda b: conv_relu_forward(x, w, b,␣
 ↪conv_param)[0], b, dout)

# Relative errors should be around e-8 or less
print('Testing conv_relu:')
print('dx error: ', rel_error(dx_num, dx))
print('dw error: ', rel_error(dw_num, dw))
print('db error: ', rel_error(db_num, db))
```

```
Testing conv_relu:
dx error:  3.5600610115232832e-09
dw error:  2.2497700915729298e-10
db error:  1.3087619975802167e-10
```

9 Three-layer ConvNet

Now that you have implemented all the necessary layers, we can put them together into a simple convolutional network.

Open the file `cs231n/classifiers/cnn.py` and complete the implementation of the `ThreeLayerConvNet` class. Remember you can use the fast/sandwich layers (already imported for you) in your implementation. Run the following cells to help you debug:

9.1 Sanity check loss

After you build a new network, one of the first things you should do is sanity check the loss. When we use the softmax loss, we expect the loss for random weights (and no regularization) to be about `log(C)` for `C` classes. When we add regularization this should go up.

[15]:
```
model = ThreeLayerConvNet()

N = 50
X = np.random.randn(N, 3, 32, 32)
y = np.random.randint(10, size=N)

loss, grads = model.loss(X, y)
print('Initial loss (no regularization): ', loss)

model.reg = 0.5
loss, grads = model.loss(X, y)
print('Initial loss (with regularization): ', loss)
```

```
Initial loss (no regularization):  2.302586071243987
Initial loss (with regularization):  2.508255638232932
```

9.2 Gradient check

After the loss looks reasonable, use numeric gradient checking to make sure that your backward pass is correct. When you use numeric gradient checking you should use a small amount of artifical data and a small number of neurons at each layer. Note: correct implementations may still have relative errors up to the order of e-2.

[16]:
```
num_inputs = 2
input_dim = (3, 16, 16)
reg = 0.0
num_classes = 10
np.random.seed(231)
X = np.random.randn(num_inputs, *input_dim)
y = np.random.randint(num_classes, size=num_inputs)

model = ThreeLayerConvNet(num_filters=3, filter_size=3,
                          input_dim=input_dim, hidden_dim=7,
                          dtype=np.float64)
loss, grads = model.loss(X, y)
# Errors should be small, but correct implementations may have
# relative errors up to the order of e-2
for param_name in sorted(grads):
    f = lambda _: model.loss(X, y)[0]
    param_grad_num = eval_numerical_gradient(f, model.params[param_name],␣
 ↪verbose=False, h=1e-6)
    e = rel_error(param_grad_num, grads[param_name])
    print('%s max relative error: %e' % (param_name, rel_error(param_grad_num,␣
 ↪grads[param_name])))
```

```
W1 max relative error: 1.380104e-04
W2 max relative error: 1.822723e-02
W3 max relative error: 3.064049e-04
```

```
b1 max relative error: 3.477652e-05
b2 max relative error: 2.516375e-03
b3 max relative error: 7.945660e-10
```

9.3 Overfit small data

A nice trick is to train your model with just a few training samples. You should be able to overfit small datasets, which will result in very high training accuracy and comparatively low validation accuracy.

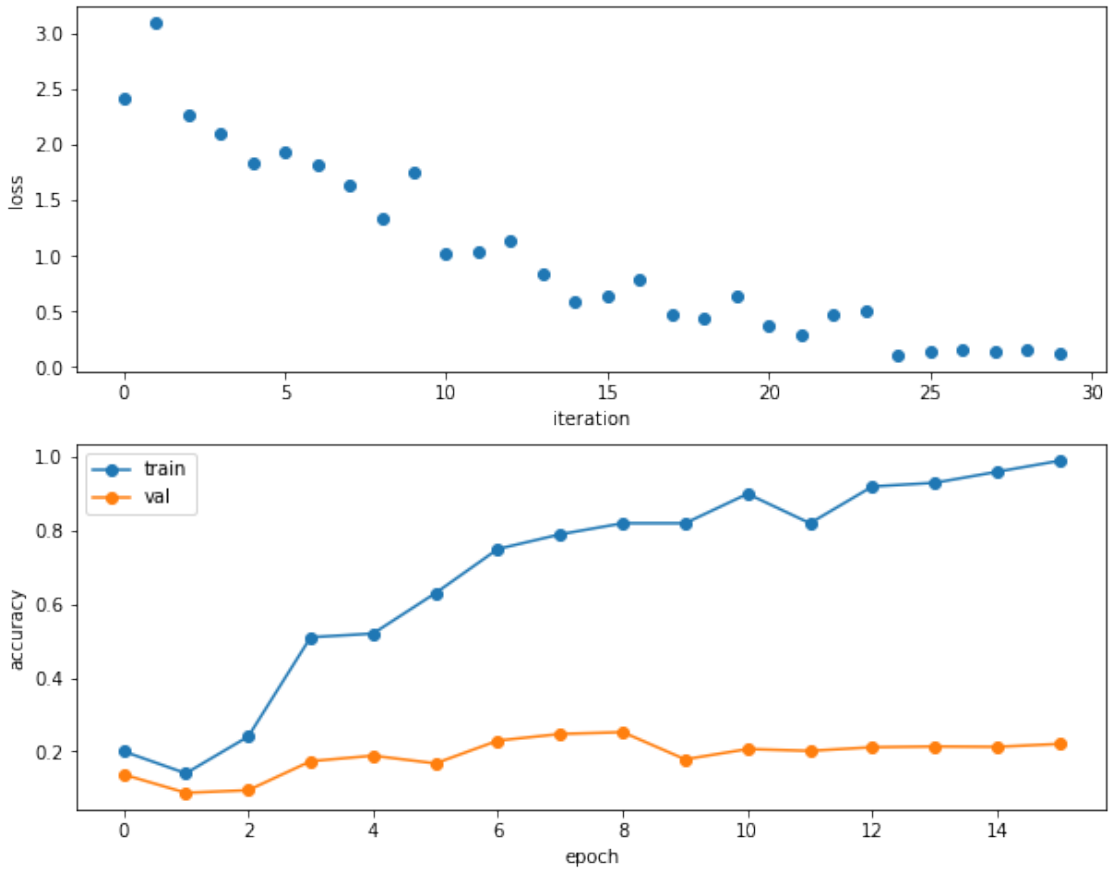
[17]:
```
np.random.seed(231)

num_train = 100
small_data = {
  'X_train': data['X_train'][:num_train],
  'y_train': data['y_train'][:num_train],
  'X_val': data['X_val'],
  'y_val': data['y_val'],
}

model = ThreeLayerConvNet(weight_scale=1e-2)

solver = Solver(model, small_data,
                num_epochs=15, batch_size=50,
                update_rule='adam',
                optim_config={
                  'learning_rate': 1e-3,
                },
                verbose=True, print_every=1)
solver.train()
```

```
(Iteration 1 / 30) loss: 2.414060
(Epoch 0 / 15) train acc: 0.200000; val_acc: 0.137000
(Iteration 2 / 30) loss: 3.102925
(Epoch 1 / 15) train acc: 0.140000; val_acc: 0.087000
(Iteration 3 / 30) loss: 2.270330
(Iteration 4 / 30) loss: 2.096705
(Epoch 2 / 15) train acc: 0.240000; val_acc: 0.094000
(Iteration 5 / 30) loss: 1.838880
(Iteration 6 / 30) loss: 1.934188
(Epoch 3 / 15) train acc: 0.510000; val_acc: 0.173000
(Iteration 7 / 30) loss: 1.827912
(Iteration 8 / 30) loss: 1.639574
(Epoch 4 / 15) train acc: 0.520000; val_acc: 0.188000
(Iteration 9 / 30) loss: 1.330082
(Iteration 10 / 30) loss: 1.756115
(Epoch 5 / 15) train acc: 0.630000; val_acc: 0.167000
(Iteration 11 / 30) loss: 1.024162
```

```
(Iteration 12 / 30) loss: 1.041826
(Epoch 6 / 15) train acc: 0.750000; val_acc: 0.229000
(Iteration 13 / 30) loss: 1.142777
(Iteration 14 / 30) loss: 0.835706
(Epoch 7 / 15) train acc: 0.790000; val_acc: 0.247000
(Iteration 15 / 30) loss: 0.587786
(Iteration 16 / 30) loss: 0.645509
(Epoch 8 / 15) train acc: 0.820000; val_acc: 0.252000
(Iteration 17 / 30) loss: 0.786844
(Iteration 18 / 30) loss: 0.467054
(Epoch 9 / 15) train acc: 0.820000; val_acc: 0.178000
(Iteration 19 / 30) loss: 0.429880
(Iteration 20 / 30) loss: 0.635498
(Epoch 10 / 15) train acc: 0.900000; val_acc: 0.206000
(Iteration 21 / 30) loss: 0.365807
(Iteration 22 / 30) loss: 0.284220
(Epoch 11 / 15) train acc: 0.820000; val_acc: 0.201000
(Iteration 23 / 30) loss: 0.469343
(Iteration 24 / 30) loss: 0.509369
(Epoch 12 / 15) train acc: 0.920000; val_acc: 0.211000
(Iteration 25 / 30) loss: 0.111638
(Iteration 26 / 30) loss: 0.145388
(Epoch 13 / 15) train acc: 0.930000; val_acc: 0.213000
(Iteration 27 / 30) loss: 0.155575
(Iteration 28 / 30) loss: 0.143398
(Epoch 14 / 15) train acc: 0.960000; val_acc: 0.212000
(Iteration 29 / 30) loss: 0.158160
(Iteration 30 / 30) loss: 0.118934
(Epoch 15 / 15) train acc: 0.990000; val_acc: 0.220000
```

Plotting the loss, training accuracy, and validation accuracy should show clear overfitting:

[18]:
```
plt.subplot(2, 1, 1)
plt.plot(solver.loss_history, 'o')
plt.xlabel('iteration')
plt.ylabel('loss')

plt.subplot(2, 1, 2)
plt.plot(solver.train_acc_history, '-o')
plt.plot(solver.val_acc_history, '-o')
plt.legend(['train', 'val'], loc='upper left')
plt.xlabel('epoch')
plt.ylabel('accuracy')
plt.show()
```



9.4 Train the net

By training the three-layer convolutional network for one epoch, you should achieve greater than 40% accuracy on the training set:

[19]:
```
model = ThreeLayerConvNet(weight_scale=0.001, hidden_dim=500, reg=0.001)

solver = Solver(model, data,
                num_epochs=1, batch_size=50,
                update_rule='adam',
                optim_config={
                  'learning_rate': 1e-3,
                },
                verbose=True, print_every=20)
solver.train()
```

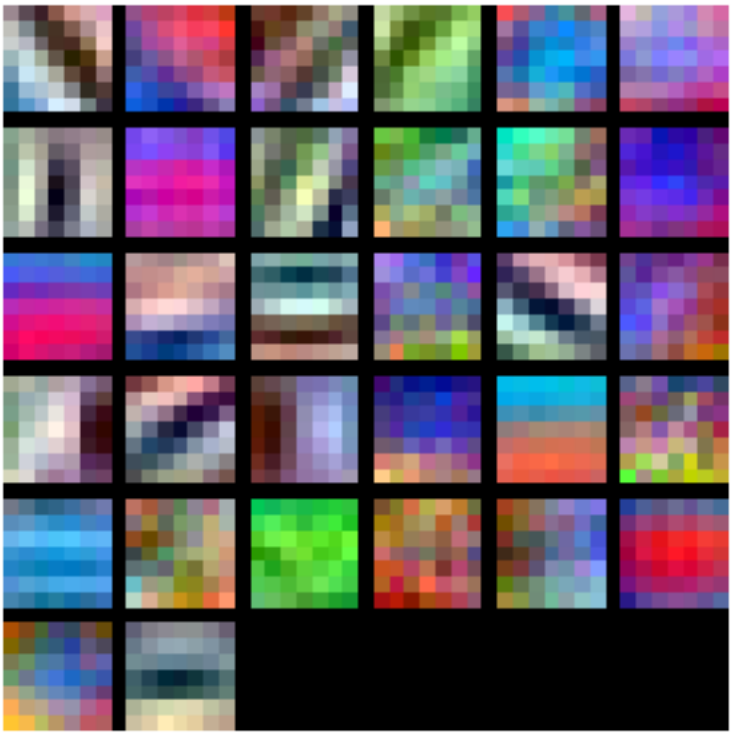
```
(Iteration 1 / 980) loss: 2.304740
(Epoch 0 / 1) train acc: 0.103000; val_acc: 0.107000
(Iteration 21 / 980) loss: 2.098229
(Iteration 41 / 980) loss: 1.949788
```

```
(Iteration 61 / 980) loss: 1.888398
(Iteration 81 / 980) loss: 1.877093
(Iteration 101 / 980) loss: 1.851877
(Iteration 121 / 980) loss: 1.859353
(Iteration 141 / 980) loss: 1.800181
(Iteration 161 / 980) loss: 2.143292
(Iteration 181 / 980) loss: 1.830573
(Iteration 201 / 980) loss: 2.037280
(Iteration 221 / 980) loss: 2.020304
(Iteration 241 / 980) loss: 1.823728
(Iteration 261 / 980) loss: 1.692679
(Iteration 281 / 980) loss: 1.882594
(Iteration 301 / 980) loss: 1.798261
(Iteration 321 / 980) loss: 1.851960
(Iteration 341 / 980) loss: 1.716323
(Iteration 361 / 980) loss: 1.897655
(Iteration 381 / 980) loss: 1.319744
(Iteration 401 / 980) loss: 1.738790
(Iteration 421 / 980) loss: 1.488866
(Iteration 441 / 980) loss: 1.718409
(Iteration 461 / 980) loss: 1.744440
(Iteration 481 / 980) loss: 1.605460
(Iteration 501 / 980) loss: 1.494847
(Iteration 521 / 980) loss: 1.835179
(Iteration 541 / 980) loss: 1.483923
(Iteration 561 / 980) loss: 1.676871
(Iteration 581 / 980) loss: 1.438325
(Iteration 601 / 980) loss: 1.443469
(Iteration 621 / 980) loss: 1.529369
(Iteration 641 / 980) loss: 1.763475
(Iteration 661 / 980) loss: 1.790329
(Iteration 681 / 980) loss: 1.693343
(Iteration 701 / 980) loss: 1.637078
(Iteration 721 / 980) loss: 1.644564
(Iteration 741 / 980) loss: 1.708919
(Iteration 761 / 980) loss: 1.494252
(Iteration 781 / 980) loss: 1.901751
(Iteration 801 / 980) loss: 1.898991
(Iteration 821 / 980) loss: 1.489988
(Iteration 841 / 980) loss: 1.377615
(Iteration 861 / 980) loss: 1.763751
(Iteration 881 / 980) loss: 1.540284
(Iteration 901 / 980) loss: 1.525582
(Iteration 921 / 980) loss: 1.674166
(Iteration 941 / 980) loss: 1.714316
(Iteration 961 / 980) loss: 1.534668
(Epoch 1 / 1) train acc: 0.504000; val_acc: 0.499000
```

9.5 Visualize Filters

You can visualize the first-layer convolutional filters from the trained network by running the following:

[20]:
```
from cs231n.vis_utils import visualize_grid

grid = visualize_grid(model.params['W1'].transpose(0, 2, 3, 1))
plt.imshow(grid.astype('uint8'))
plt.axis('off')
plt.gcf().set_size_inches(5, 5)
plt.show()
```



10 Spatial Batch Normalization

We already saw that batch normalization is a very useful technique for training deep fully-connected networks. As proposed in the original paper [3], batch normalization can also be used for convolutional networks, but we need to tweak it a bit; the modification will be called "spatial batch normalization."

Normally batch-normalization accepts inputs of shape `(N, D)` and produces outputs of shape `(N, D)`, where we normalize across the minibatch dimension `N`. For data coming from convolutional layers, batch normalization needs to accept inputs of shape `(N, C, H, W)` and produce outputs of

shape `(N, C, H, W)` where the `N` dimension gives the minibatch size and the `(H, W)` dimensions give the spatial size of the feature map.

If the feature map was produced using convolutions, then we expect the statistics of each feature channel to be relatively consistent both between different images and different locations within the same image. Therefore spatial batch normalization computes a mean and variance for each of the `C` feature channels by computing statistics over both the minibatch dimension `N` and the spatial dimensions `H` and `W`.

[3] Sergey Ioffe and Christian Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift", ICML 2015.

10.1 Spatial batch normalization: forward

In the file `cs231n/layers.py`, implement the forward pass for spatial batch normalization in the function `spatial_batchnorm_forward`. Check your implementation by running the following:

[21]:
```
np.random.seed(231)
# Check the training-time forward pass by checking means and variances
# of features both before and after spatial batch normalization

N, C, H, W = 2, 3, 4, 5
x = 4 * np.random.randn(N, C, H, W) + 10

print('Before spatial batch normalization:')
print('  Shape: ', x.shape)
print('  Means: ', x.mean(axis=(0, 2, 3)))
print('  Stds: ', x.std(axis=(0, 2, 3)))

# Means should be close to zero and stds close to one
gamma, beta = np.ones(C), np.zeros(C)
bn_param = {'mode': 'train'}
out, _ = spatial_batchnorm_forward(x, gamma, beta, bn_param)
print('After spatial batch normalization:')
print('  Shape: ', out.shape)
print('  Means: ', out.mean(axis=(0, 2, 3)))
print('  Stds: ', out.std(axis=(0, 2, 3)))

# Means should be close to beta and stds close to gamma
gamma, beta = np.asarray([3, 4, 5]), np.asarray([6, 7, 8])
out, _ = spatial_batchnorm_forward(x, gamma, beta, bn_param)
print('After spatial batch normalization (nontrivial gamma, beta):')
print('  Shape: ', out.shape)
print('  Means: ', out.mean(axis=(0, 2, 3)))
print('  Stds: ', out.std(axis=(0, 2, 3)))
```

```
Before spatial batch normalization:
  Shape:  (2, 3, 4, 5)
  Means:  [9.33463814 8.90909116 9.11056338]
```

```
  Stds:  [3.61447857 3.19347686 3.5168142 ]
After spatial batch normalization:
  Shape:  (2, 3, 4, 5)
  Means:  [ 5.85642645e-16  5.93969318e-16 -8.88178420e-17]
  Stds:  [0.99999962 0.99999951 0.9999996 ]
After spatial batch normalization (nontrivial gamma, beta):
  Shape:  (2, 3, 4, 5)
  Means:  [6. 7. 8.]
  Stds:  [2.99999885 3.99999804 4.99999798]
```

[22]:
```
np.random.seed(231)
# Check the test-time forward pass by running the training-time
# forward pass many times to warm up the running averages, and then
# checking the means and variances of activations after a test-time
# forward pass.
N, C, H, W = 10, 4, 11, 12

bn_param = {'mode': 'train'}
gamma = np.ones(C)
beta = np.zeros(C)
for t in range(50):
  x = 2.3 * np.random.randn(N, C, H, W) + 13
  spatial_batchnorm_forward(x, gamma, beta, bn_param)
bn_param['mode'] = 'test'
x = 2.3 * np.random.randn(N, C, H, W) + 13
a_norm, _ = spatial_batchnorm_forward(x, gamma, beta, bn_param)

# Means should be close to zero and stds close to one, but will be
# noisier than training-time forward passes.
print('After spatial batch normalization (test-time):')
print('  means: ', a_norm.mean(axis=(0, 2, 3)))
print('  stds: ', a_norm.std(axis=(0, 2, 3)))
```

```
After spatial batch normalization (test-time):
  means:  [-0.08034406  0.07562881  0.05716371  0.04378383]
  stds:  [0.96718744 1.0299714  1.02887624 1.00585577]
```

10.2 Spatial batch normalization: backward

In the file `cs231n/layers.py`, implement the backward pass for spatial batch normalization in the function `spatial_batchnorm_backward`. Run the following to check your implementation using a numeric gradient check:

[23]:
```
np.random.seed(231)
N, C, H, W = 2, 3, 4, 5
x = 5 * np.random.randn(N, C, H, W) + 12
gamma = np.random.randn(C)
beta = np.random.randn(C)
```

```
dout = np.random.randn(N, C, H, W)

bn_param = {'mode': 'train'}
fx = lambda x: spatial_batchnorm_forward(x, gamma, beta, bn_param)[0]
fg = lambda a: spatial_batchnorm_forward(x, gamma, beta, bn_param)[0]
fb = lambda b: spatial_batchnorm_forward(x, gamma, beta, bn_param)[0]

dx_num = eval_numerical_gradient_array(fx, x, dout)
da_num = eval_numerical_gradient_array(fg, gamma, dout)
db_num = eval_numerical_gradient_array(fb, beta, dout)

#You should expect errors of magnitudes between 1e-12~1e-06
_, cache = spatial_batchnorm_forward(x, gamma, beta, bn_param)
dx, dgamma, dbeta = spatial_batchnorm_backward(dout, cache)
print('dx error: ', rel_error(dx_num, dx))
print('dgamma error: ', rel_error(da_num, dgamma))
print('dbeta error: ', rel_error(db_num, dbeta))
```

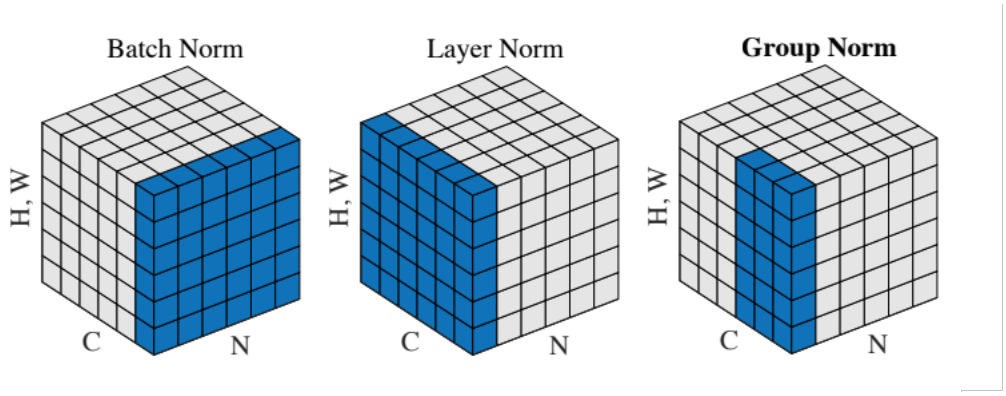
```
dx error:  3.083846820796372e-07
dgamma error:  7.09738489671469e-12
dbeta error:  3.275608725278405e-12
```

11 Group Normalization

In the previous notebook, we mentioned that Layer Normalization is an alternative normalization technique that mitigates the batch size limitations of Batch Normalization. However, as the authors of [4] observed, Layer Normalization does not perform as well as Batch Normalization when used with Convolutional Layers:

> With fully connected layers, all the hidden units in a layer tend to make similar contributions to the final prediction, and re-centering and rescaling the summed inputs to a layer works well. However, the assumption of similar contributions is no longer true for convolutional neural networks. The large number of the hidden units whose receptive fields lie near the boundary of the image are rarely turned on and thus have very different statistics from the rest of the hidden units within the same layer.

The authors of [5] propose an intermediary technique. In contrast to Layer Normalization, where you normalize over the entire feature per-datapoint, they suggest a consistent splitting of each per-datapoint feature into G groups, and a per-group per-datapoint normalization instead.



Visual comparison of the normalization techniques discussed so far (image edited from [5])

Even though an assumption of equal contribution is still being made within each group, the authors hypothesize that this is not as problematic, as innate grouping arises within features for visual recognition. One example they use to illustrate this is that many high-performance handcrafted features in traditional Computer Vision have terms that are explicitly grouped together. Take for example Histogram of Oriented Gradients [6]– after computing histograms per spatially local block, each per-block histogram is normalized before being concatenated together to form the final feature vector.

You will now implement Group Normalization. Note that this normalization technique that you are to implement in the following cells was introduced and published to arXiv *less than a month ago* – this truly is still an ongoing and excitingly active field of research!

[4] Ba, Jimmy Lei, Jamie Ryan Kiros, and Geoffrey E. Hinton. "Layer Normalization." stat 1050 (2016): 21.

[5] Wu, Yuxin, and Kaiming He. "Group Normalization." arXiv preprint arXiv:1803.08494 (2018).

[6] N. Dalal and B. Triggs. Histograms of oriented gradients for human detection. In Computer Vision and Pattern Recognition (CVPR), 2005.

11.1 Group normalization: forward

In the file `cs231n/layers.py`, implement the forward pass for group normalization in the function `spatial_groupnorm_forward`. Check your implementation by running the following:

[24]:
```
np.random.seed(231)
# Check the training-time forward pass by checking means and variances
# of features both before and after spatial batch normalization

N, C, H, W = 2, 6, 4, 5
G = 2
x = 4 * np.random.randn(N, C, H, W) + 10
x_g = x.reshape((N*G,-1))
print('Before spatial group normalization:')
print('  Shape: ', x.shape)
print('  Means: ', x_g.mean(axis=1))
print('  Stds: ', x_g.std(axis=1))
```

```
# Means should be close to zero and stds close to one
gamma, beta = np.ones((1,C,1,1)), np.zeros((1,C,1,1))
bn_param = {'mode': 'train'}

out, _ = spatial_groupnorm_forward(x, gamma, beta, G, bn_param)
out_g = out.reshape((N*G,-1))
print('After spatial group normalization:')
print('  Shape: ', out.shape)
print('  Means: ', out_g.mean(axis=1))
print('  Stds: ', out_g.std(axis=1))
```

```
Before spatial group normalization:
  Shape:  (2, 6, 4, 5)
  Means:  [9.72505327 8.51114185 8.9147544  9.43448077]
  Stds:  [3.67070958 3.09892597 4.27043622 3.97521327]
After spatial group normalization:
  Shape:  (1, 1, 1, 2, 6, 4, 5)
  Means:  [-2.14643118e-16  5.25505565e-16  2.58126853e-16 -3.62672855e-16]
  Stds:  [0.99999963 0.99999948 0.99999973 0.99999968]
```

11.2 Spatial group normalization: backward

In the file `cs231n/layers.py`, implement the backward pass for spatial batch normalization in the function `spatial_groupnorm_backward`. Run the following to check your implementation using a numeric gradient check:

[25]:
```
np.random.seed(231)
N, C, H, W = 2, 6, 4, 5
G = 2
x = 5 * np.random.randn(N, C, H, W) + 12
gamma = np.random.randn(1,C,1,1)
beta = np.random.randn(1,C,1,1)
dout = np.random.randn(N, C, H, W)

gn_param = {}
fx = lambda x: spatial_groupnorm_forward(x, gamma, beta, G, gn_param)[0]
fg = lambda a: spatial_groupnorm_forward(x, gamma, beta, G, gn_param)[0]
fb = lambda b: spatial_groupnorm_forward(x, gamma, beta, G, gn_param)[0]

dx_num = eval_numerical_gradient_array(fx, x, dout)
da_num = eval_numerical_gradient_array(fg, gamma, dout)
db_num = eval_numerical_gradient_array(fb, beta, dout)

_, cache = spatial_groupnorm_forward(x, gamma, beta, G, gn_param)
dx, dgamma, dbeta = spatial_groupnorm_backward(dout, cache)
#You should expect errors of magnitudes between 1e-12~1e-07
```

```
print('dx error: ', rel_error(dx_num, dx))
print('dgamma error: ', rel_error(da_num, dgamma))
print('dbeta error: ', rel_error(db_num, dbeta))
```

```
dx error:  6.34590431845254e-08
dgamma error:  1.0546047434202244e-11
dbeta error:  3.810857316122484e-12
```

5 Output of PyTorch.ipynb

The following pages in this section contain the output of the file ”PyTorch.ipynb”.

1 What's this PyTorch business?

You've written a lot of code in this assignment to provide a whole host of neural network functionality. Dropout, Batch Norm, and 2D convolutions are some of the workhorses of deep learning in computer vision. You've also worked hard to make your code efficient and vectorized.

For the last part of this assignment, though, we're going to leave behind your beautiful codebase and instead migrate to one of two popular deep learning frameworks: in this instance, PyTorch (or TensorFlow, if you switch over to that notebook).

1.0.1 What is PyTorch?

PyTorch is a system for executing dynamic computational graphs over Tensor objects that behave similarly as numpy ndarray. It comes with a powerful automatic differentiation engine that removes the need for manual back-propagation.

1.0.2 Why?

- Our code will now run on GPUs! Much faster training. When using a framework like PyTorch or TensorFlow you can harness the power of the GPU for your own custom neural network architectures without having to write CUDA code directly (which is beyond the scope of this class).
- We want you to be ready to use one of these frameworks for your project so you can experiment more efficiently than if you were writing every feature you want to use by hand.
- We want you to stand on the shoulders of giants! TensorFlow and PyTorch are both excellent frameworks that will make your lives a lot easier, and now that you understand their guts, you are free to use them :)
- We want you to be exposed to the sort of deep learning code you might run into in academia or industry.

1.0.3 PyTorch versions

This notebook assumes that you are using **PyTorch version 0.4**. Prior to this version, Tensors had to be wrapped in Variable objects to be used in autograd; however Variables have now been deprecated. In addition 0.4 also separates a Tensor's datatype from its device, and uses numpy-style factories for constructing Tensors rather than directly invoking Tensor constructors.

1.1 How will I learn PyTorch?

Justin Johnson has made an excellent tutorial for PyTorch.

You can also find the detailed API doc here. If you have other questions that are not addressed by the API docs, the PyTorch forum is a much better place to ask than StackOverflow.

2 Table of Contents

This assignment has 5 parts. You will learn PyTorch on different levels of abstractions, which will help you understand it better and prepare you for the final project.

1. Preparation: we will use CIFAR-10 dataset.

2. Barebones PyTorch: we will work directly with the lowest-level PyTorch Tensors.
3. PyTorch Module API: we will use `nn.Module` to define arbitrary neural network architecture.
4. PyTorch Sequential API: we will use `nn.Sequential` to define a linear feed-forward network very conveniently.
5. CIFAR-10 open-ended challenge: please implement your own network to get as high accuracy as possible on CIFAR-10. You can experiment with any layer, optimizer, hyperparameters or other advanced features.

Here is a table of comparison:

API	Flexibility	Convenience
Barebone	High	Low
`nn.Module`	High	Medium
`nn.Sequential`	Low	High

3 Part I. Preparation

First, we load the CIFAR-10 dataset. This might take a couple minutes the first time you do it, but the files should stay cached after that.

In previous parts of the assignment we had to write our own code to download the CIFAR-10 dataset, preprocess it, and iterate through it in minibatches; PyTorch provides convenient tools to automate this process for us.

[36]:
```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader
from torch.utils.data import sampler

import torchvision.datasets as dset
import torchvision.transforms as T

import numpy as np
```

[37]:
```
NUM_TRAIN = 49000

# The torchvision.transforms package provides tools for preprocessing data
# and for performing data augmentation; here we set up a transform to
# preprocess the data by subtracting the mean RGB value and dividing by the
# standard deviation of each RGB value; we've hardcoded the mean and std.
transform = T.Compose([
                T.ToTensor(),
                T.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010))
            ])

# We set up a Dataset object for each split (train / val / test); Datasets load
```

```
# training examples one at a time, so we wrap each Dataset in a DataLoader which
# iterates through the Dataset and forms minibatches. We divide the CIFAR-10
# training set into train and val sets by passing a Sampler object to the
# DataLoader telling how it should sample from the underlying Dataset.
cifar10_train = dset.CIFAR10('./cs231n/datasets', train=True, download=True,
                             transform=transform)
loader_train = DataLoader(cifar10_train, batch_size=64,
                          sampler=sampler.SubsetRandomSampler(range(NUM_TRAIN)))

cifar10_val = dset.CIFAR10('./cs231n/datasets', train=True, download=True,
                           transform=transform)
loader_val = DataLoader(cifar10_val, batch_size=64,
                        sampler=sampler.SubsetRandomSampler(range(NUM_TRAIN,␣
↪50000)))

cifar10_test = dset.CIFAR10('./cs231n/datasets', train=False, download=True,
                            transform=transform)
loader_test = DataLoader(cifar10_test, batch_size=64)
```

```
Files already downloaded and verified
Files already downloaded and verified
Files already downloaded and verified
```

You have an option to **use GPU by setting the flag to True below**. It is not necessary to use GPU for this assignment. Note that if your computer does not have CUDA enabled, `torch.cuda.is_available()` will return False and this notebook will fallback to CPU mode.

The global variables `dtype` and `device` will control the data types throughout this assignment.

[38]:
```
USE_GPU = True

dtype = torch.float32 # we will be using float throughout this tutorial

if USE_GPU and torch.cuda.is_available():
    device = torch.device('cuda')
else:
    device = torch.device('cpu')

# Constant to control how frequently we print train loss
print_every = 100

print('using device:', device)
```

```
using device: cpu
```

4 Part II. Barebones PyTorch

PyTorch ships with high-level APIs to help us define model architectures conveniently, which we will cover in Part II of this tutorial. In this section, we will start with the barebone PyTorch elements to understand the autograd engine better. After this exercise, you will come to appreciate the high-level model API more.

We will start with a simple fully-connected ReLU network with two hidden layers and no biases for CIFAR classification. This implementation computes the forward pass using operations on PyTorch Tensors, and uses PyTorch autograd to compute gradients. It is important that you understand every line, because you will write a harder version after the example.

When we create a PyTorch Tensor with `requires_grad=True`, then operations involving that Tensor will not just compute values; they will also build up a computational graph in the background, allowing us to easily backpropagate through the graph to compute gradients of some Tensors with respect to a downstream loss. Concretely if x is a Tensor with `x.requires_grad == True` then after backpropagation `x.grad` will be another Tensor holding the gradient of x with respect to the scalar loss at the end.

4.0.1 PyTorch Tensors: Flatten Function

A PyTorch Tensor is conceptionally similar to a numpy array: it is an n-dimensional grid of numbers, and like numpy PyTorch provides many functions to efficiently operate on Tensors. As a simple example, we provide a `flatten` function below which reshapes image data for use in a fully-connected neural network.

Recall that image data is typically stored in a Tensor of shape N x C x H x W, where:

- N is the number of datapoints
- C is the number of channels
- H is the height of the intermediate feature map in pixels
- W is the height of the intermediate feature map in pixels

This is the right way to represent the data when we are doing something like a 2D convolution, that needs spatial understanding of where the intermediate features are relative to each other. When we use fully connected affine layers to process the image, however, we want each datapoint to be represented by a single vector – it's no longer useful to segregate the different channels, rows, and columns of the data. So, we use a "flatten" operation to collapse the `C x H x W` values per representation into a single long vector. The flatten function below first reads in the N, C, H, and W values from a given batch of data, and then returns a "view" of that data. "View" is analogous to numpy's "reshape" method: it reshapes x's dimensions to be N x ??, where ?? is allowed to be anything (in this case, it will be C x H x W, but we don't need to specify that explicitly).

[39]:
```
def flatten(x):
    N = x.shape[0] # read in N, C, H, W
    return x.view(N, -1)  # "flatten" the C * H * W values into a single vector␣
 ↪per image

def test_flatten():
    x = torch.arange(12).view(2, 1, 3, 2)
```

```
    print('Before flattening: ', x)
    print('After flattening: ', flatten(x))

test_flatten()
```

```
Before flattening:  tensor([[[[ 0,  1],
          [ 2,  3],
          [ 4,  5]]],


        [[[ 6,  7],
          [ 8,  9],
          [10, 11]]]])
After flattening:  tensor([[ 0,  1,  2,  3,  4,  5],
        [ 6,  7,  8,  9, 10, 11]])
```

4.0.2 Barebones PyTorch: Two-Layer Network

Here we define a function `two_layer_fc` which performs the forward pass of a two-layer fully-connected ReLU network on a batch of image data. After defining the forward pass we check that it doesn't crash and that it produces outputs of the right shape by running zeros through the network.

You don't have to write any code here, but it's important that you read and understand the implementation.

[40]:
```
import torch.nn.functional as F  # useful stateless functions

def two_layer_fc(x, params):
    """
    A fully-connected neural networks; the architecture is:
    NN is fully connected -> ReLU -> fully connected layer.
    Note that this function only defines the forward pass;
    PyTorch will take care of the backward pass for us.

    The input to the network will be a minibatch of data, of shape
    (N, d1, ..., dM) where d1 * ... * dM = D. The hidden layer will have H units,
    and the output layer will produce scores for C classes.

    Inputs:
    - x: A PyTorch Tensor of shape (N, d1, ..., dM) giving a minibatch of
      input data.
    - params: A list [w1, w2] of PyTorch Tensors giving weights for the network;
      w1 has shape (D, H) and w2 has shape (H, C).

    Returns:
    - scores: A PyTorch Tensor of shape (N, C) giving classification scores for
      the input data x.
```

```
    """
    # first we flatten the image
    x = flatten(x)  # shape: [batch_size, C x H x W]

    w1, w2 = params

    # Forward pass: compute predicted y using operations on Tensors. Since w1 and
    # w2 have requires_grad=True, operations involving these Tensors will cause
    # PyTorch to build a computational graph, allowing automatic computation of
    # gradients. Since we are no longer implementing the backward pass by hand we
    # don't need to keep references to intermediate values.
    # you can also use `.clamp(min=0)`, equivalent to F.relu()
    x = F.relu(x.mm(w1))
    x = x.mm(w2)
    return x


def two_layer_fc_test():
    hidden_layer_size = 42
    x = torch.zeros((64, 50), dtype=dtype)  # minibatch size 64, feature␣
 ↪dimension 50
    w1 = torch.zeros((50, hidden_layer_size), dtype=dtype)
    w2 = torch.zeros((hidden_layer_size, 10), dtype=dtype)
    scores = two_layer_fc(x, [w1, w2])
    print(scores.size())  # you should see [64, 10]

two_layer_fc_test()
```

```
torch.Size([64, 10])
```

4.0.3 Barebones PyTorch: Three-Layer ConvNet

Here you will complete the implementation of the function `three_layer_convnet`, which will perform the forward pass of a three-layer convolutional network. Like above, we can immediately test our implementation by passing zeros through the network. The network should have the following architecture:

1. A convolutional layer (with bias) with `channel_1` filters, each with shape `KW1 x KH1`, and zero-padding of two
2. ReLU nonlinearity
3. A convolutional layer (with bias) with `channel_2` filters, each with shape `KW2 x KH2`, and zero-padding of one
4. ReLU nonlinearity
5. Fully-connected layer with bias, producing scores for C classes.

HINT: For convolutions: http://pytorch.org/docs/stable/nn.html#torch.nn.functional.conv2d; pay attention to the shapes of convolutional filters!

[41]:
```
def three_layer_convnet(x, params):
    """
    Performs the forward pass of a three-layer convolutional network with the
    architecture defined above.

    Inputs:
    - x: A PyTorch Tensor of shape (N, 3, H, W) giving a minibatch of images
    - params: A list of PyTorch Tensors giving the weights and biases for the
      network; should contain the following:
      - conv_w1: PyTorch Tensor of shape (channel_1, 3, KH1, KW1) giving weights
        for the first convolutional layer
      - conv_b1: PyTorch Tensor of shape (channel_1,) giving biases for the first
        convolutional layer
      - conv_w2: PyTorch Tensor of shape (channel_2, channel_1, KH2, KW2) giving
        weights for the second convolutional layer
      - conv_b2: PyTorch Tensor of shape (channel_2,) giving biases for the␣
↪second
        convolutional layer
      - fc_w: PyTorch Tensor giving weights for the fully-connected layer. Can␣
↪you
        figure out what the shape should be?
      - fc_b: PyTorch Tensor giving biases for the fully-connected layer. Can you
        figure out what the shape should be?

    Returns:
    - scores: PyTorch Tensor of shape (N, C) giving classification scores for x
    """
    conv_w1, conv_b1, conv_w2, conv_b2, fc_w, fc_b = params
    scores = None
    ␣
↪################################################################################
    # TODO: Implement the forward pass for the three-layer ConvNet.             ␣
↪  #
    ␣
↪################################################################################

    conv1 = F.conv2d(x, weight=conv_w1, bias=conv_b1, padding=2)
    relu1 = F.relu(conv1)
    conv2 = F.conv2d(relu1, weight=conv_w2, bias=conv_b2, padding=1)
    relu2 = F.relu(conv2)
    relu2_flat = flatten(relu2)
    scores = relu2_flat.mm(fc_w) + fc_b

    ␣
↪################################################################################
    #                                 END OF YOUR CODE                          ␣
↪  #
```

```
␣
↪##############################################################################
    return scores
```

After defining the forward pass of the ConvNet above, run the following cell to test your implementation.

When you run this function, scores should have shape (64, 10).

[42]:
```
def three_layer_convnet_test():
    x = torch.zeros((64, 3, 32, 32), dtype=dtype)  # minibatch size 64, image␣
↪size [3, 32, 32]

    conv_w1 = torch.zeros((6, 3, 5, 5), dtype=dtype)  # [out_channel,␣
↪in_channel, kernel_H, kernel_W]
    conv_b1 = torch.zeros((6,))  # out_channel
    conv_w2 = torch.zeros((9, 6, 3, 3), dtype=dtype)  # [out_channel,␣
↪in_channel, kernel_H, kernel_W]
    conv_b2 = torch.zeros((9,))  # out_channel

    # you must calculate the shape of the tensor after two conv layers, before␣
↪the fully-connected layer
    fc_w = torch.zeros((9 * 32 * 32, 10))
    fc_b = torch.zeros(10)

    scores = three_layer_convnet(x, [conv_w1, conv_b1, conv_w2, conv_b2, fc_w,␣
↪fc_b])
    print(scores.size())  # you should see [64, 10]
three_layer_convnet_test()
```

```
torch.Size([64, 10])
```

4.0.4 Barebones PyTorch: Initialization

Let's write a couple utility methods to initialize the weight matrices for our models.

- `random_weight(shape)` initializes a weight tensor with the Kaiming normalization method.
- `zero_weight(shape)` initializes a weight tensor with all zeros. Useful for instantiating bias parameters.

The `random_weight` function uses the Kaiming normal initialization method, described in:

He et al, *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification*, ICCV 2015, https://arxiv.org/abs/1502.01852

[43]:
```
def random_weight(shape):
    """
    Create random Tensors for weights; setting requires_grad=True means that we
    want to compute gradients for these Tensors during the backward pass.
```

```
    We use Kaiming normalization: sqrt(2 / fan_in)
    """
    if len(shape) == 2:  # FC weight
        fan_in = shape[0]
    else:
        fan_in = np.prod(shape[1:]) # conv weight [out_channel, in_channel, kH, kW]
    # randn is standard normal distribution generator.
    w = torch.randn(shape, device=device, dtype=dtype) * np.sqrt(2. / fan_in)
    w.requires_grad = True
    return w

def zero_weight(shape):
    return torch.zeros(shape, device=device, dtype=dtype, requires_grad=True)

# create a weight of shape [3 x 5]
# you should see the type `torch.cuda.FloatTensor` if you use GPU.
# Otherwise it should be `torch.FloatTensor`
random_weight((3, 5))
```

[43]:
```
tensor([[ 0.1408, -0.0389, -0.4260, -0.3820, -0.0399],
        [-0.4846, -0.3228, -0.0412, -0.9957,  0.5556],
        [-0.0811,  0.1376,  0.4165, -0.2023, -0.6135]], requires_grad=True)
```

4.0.5 Barebones PyTorch: Check Accuracy

When training the model we will use the following function to check the accuracy of our model on the training or validation sets.

When checking accuracy we don't need to compute any gradients; as a result we don't need PyTorch to build a computational graph for us when we compute scores. To prevent a graph from being built we scope our computation under a `torch.no_grad()` context manager.

[44]:
```
def check_accuracy_part2(loader, model_fn, params):
    """
    Check the accuracy of a classification model.

    Inputs:
    - loader: A DataLoader for the data split we want to check
    - model_fn: A function that performs the forward pass of the model,
      with the signature scores = model_fn(x, params)
    - params: List of PyTorch Tensors giving parameters of the model

    Returns: Nothing, but prints the accuracy of the model
    """
    split = 'val' if loader.dataset.train else 'test'
    print('Checking accuracy on the %s set' % split)
```

```
    num_correct, num_samples = 0, 0
    with torch.no_grad():
        for x, y in loader:
            x = x.to(device=device, dtype=dtype)  # move to device, e.g. GPU
            y = y.to(device=device, dtype=torch.int64)
            scores = model_fn(x, params)
            _, preds = scores.max(1)
            num_correct += (preds == y).sum()
            num_samples += preds.size(0)
        acc = float(num_correct) / num_samples
        print('Got %d / %d correct (%.2f%%)' % (num_correct, num_samples, 100 *␣
↪acc))
```

4.0.6 BareBones PyTorch: Training Loop

We can now set up a basic training loop to train our network. We will train the model using stochastic gradient descent without momentum. We will use `torch.functional.cross_entropy` to compute the loss; you can read about it here.

The training loop takes as input the neural network function, a list of initialized parameters (`[w1, w2]` in our example), and learning rate.

[45]:
```
def train_part2(model_fn, params, learning_rate):
    """
    Train a model on CIFAR-10.

    Inputs:
    - model_fn: A Python function that performs the forward pass of the model.
      It should have the signature scores = model_fn(x, params) where x is a
      PyTorch Tensor of image data, params is a list of PyTorch Tensors giving
      model weights, and scores is a PyTorch Tensor of shape (N, C) giving
      scores for the elements in x.
    - params: List of PyTorch Tensors giving weights for the model
    - learning_rate: Python scalar giving the learning rate to use for SGD

    Returns: Nothing
    """
    for t, (x, y) in enumerate(loader_train):
        # Move the data to the proper device (GPU or CPU)
        x = x.to(device=device, dtype=dtype)
        y = y.to(device=device, dtype=torch.long)

        # Forward pass: compute scores and loss
        scores = model_fn(x, params)
        loss = F.cross_entropy(scores, y)

        # Backward pass: PyTorch figures out which Tensors in the computational
```

```
        # graph has requires_grad=True and uses backpropagation to compute the
        # gradient of the loss with respect to these Tensors, and stores the
        # gradients in the .grad attribute of each Tensor.
        loss.backward()

        # Update parameters. We don't want to backpropagate through the
        # parameter updates, so we scope the updates under a torch.no_grad()
        # context manager to prevent a computational graph from being built.
        with torch.no_grad():
            for w in params:
                w -= learning_rate * w.grad

                # Manually zero the gradients after running the backward pass
                w.grad.zero_()

        if t % print_every == 0:
            print('Iteration %d, loss = %.4f' % (t, loss.item()))
            check_accuracy_part2(loader_val, model_fn, params)
            print()
```

4.0.7 BareBones PyTorch: Train a Two-Layer Network

Now we are ready to run the training loop. We need to explicitly allocate tensors for the fully connected weights, `w1` and `w2`.

Each minibatch of CIFAR has 64 examples, so the tensor shape is `[64, 3, 32, 32]`.

After flattening, `x` shape should be `[64, 3 * 32 * 32]`. This will be the size of the first dimension of `w1`. The second dimension of `w1` is the hidden layer size, which will also be the first dimension of `w2`.

Finally, the output of the network is a 10-dimensional vector that represents the probability distribution over 10 classes.

You don't need to tune any hyperparameters but you should see accuracies above 40% after training for one epoch.

[46]:
```
hidden_layer_size = 4000
learning_rate = 1e-2

w1 = random_weight((3 * 32 * 32, hidden_layer_size))
w2 = random_weight((hidden_layer_size, 10))

train_part2(two_layer_fc, [w1, w2], learning_rate)
```

```
Iteration 0, loss = 3.1595
Checking accuracy on the val set
Got 145 / 1000 correct (14.50%)
```

```
Iteration 100, loss = 2.1859
Checking accuracy on the val set
Got 266 / 1000 correct (26.60%)

Iteration 200, loss = 2.0611
Checking accuracy on the val set
Got 325 / 1000 correct (32.50%)

Iteration 300, loss = 2.1126
Checking accuracy on the val set
Got 381 / 1000 correct (38.10%)

Iteration 400, loss = 2.1363
Checking accuracy on the val set
Got 354 / 1000 correct (35.40%)

Iteration 500, loss = 1.9434
Checking accuracy on the val set
Got 425 / 1000 correct (42.50%)

Iteration 600, loss = 1.9825
Checking accuracy on the val set
Got 408 / 1000 correct (40.80%)

Iteration 700, loss = 1.6099
Checking accuracy on the val set
Got 435 / 1000 correct (43.50%)
```

4.0.8 BareBones PyTorch: Training a ConvNet

In the below you should use the functions defined above to train a three-layer convolutional network on CIFAR. The network should have the following architecture:

1. Convolutional layer (with bias) with 32 5x5 filters, with zero-padding of 2
2. ReLU
3. Convolutional layer (with bias) with 16 3x3 filters, with zero-padding of 1
4. ReLU
5. Fully-connected layer (with bias) to compute scores for 10 classes

You should initialize your weight matrices using the `random_weight` function defined above, and you should initialize your bias vectors using the `zero_weight` function above.

You don't need to tune any hyperparameters, but if everything works correctly you should achieve an accuracy above 42% after one epoch.

[47]:
```
learning_rate = 3e-3

channel_1 = 32
```

```
channel_2 = 16

conv_w1 = None
conv_b1 = None
conv_w2 = None
conv_b2 = None
fc_w = None
fc_b = None

################################################################################
# TODO: Initialize the parameters of a three-layer ConvNet.                    #
################################################################################

conv_w1 = random_weight((channel_1, 3, 5, 5))
conv_b1 = zero_weight((channel_1,))
conv_w2 = random_weight((channel_2, 32, 3, 3))
conv_b2 = zero_weight((channel_2,))
fc_w = random_weight((channel_2*32*32, 10))
fc_b = zero_weight((10,))

################################################################################
#                                 END OF YOUR CODE                             #
################################################################################

params = [conv_w1, conv_b1, conv_w2, conv_b2, fc_w, fc_b]
train_part2(three_layer_convnet, params, learning_rate)
```

```
Iteration 0, loss = 4.1946
Checking accuracy on the val set
Got 110 / 1000 correct (11.00%)

Iteration 100, loss = 1.8672
Checking accuracy on the val set
Got 368 / 1000 correct (36.80%)

Iteration 200, loss = 1.8601
Checking accuracy on the val set
Got 425 / 1000 correct (42.50%)

Iteration 300, loss = 1.7302
Checking accuracy on the val set
Got 452 / 1000 correct (45.20%)

Iteration 400, loss = 1.7479
Checking accuracy on the val set
Got 441 / 1000 correct (44.10%)
```

```
Iteration 500, loss = 1.7347
Checking accuracy on the val set
Got 467 / 1000 correct (46.70%)

Iteration 600, loss = 1.5753
Checking accuracy on the val set
Got 471 / 1000 correct (47.10%)

Iteration 700, loss = 1.5783
Checking accuracy on the val set
Got 478 / 1000 correct (47.80%)
```

5 Part III. PyTorch Module API

Barebone PyTorch requires that we track all the parameter tensors by hand. This is fine for small networks with a few tensors, but it would be extremely inconvenient and error-prone to track tens or hundreds of tensors in larger networks.

PyTorch provides the `nn.Module` API for you to define arbitrary network architectures, while tracking every learnable parameters for you. In Part II, we implemented SGD ourselves. PyTorch also provides the `torch.optim` package that implements all the common optimizers, such as RMSProp, Adagrad, and Adam. It even supports approximate second-order methods like L-BFGS! You can refer to the doc for the exact specifications of each optimizer.

To use the Module API, follow the steps below:

1. Subclass `nn.Module`. Give your network class an intuitive name like `TwoLayerFC`.
2. In the constructor `__init__()`, define all the layers you need as class attributes. Layer objects like `nn.Linear` and `nn.Conv2d` are themselves `nn.Module` subclasses and contain learnable parameters, so that you don't have to instantiate the raw tensors yourself. `nn.Module` will track these internal parameters for you. Refer to the doc to learn more about the dozens of builtin layers. **Warning**: don't forget to call the `super().__init__()` first!
3. In the `forward()` method, define the *connectivity* of your network. You should use the attributes defined in `__init__` as function calls that take tensor as input and output the "transformed" tensor. Do *not* create any new layers with learnable parameters in `forward()`! All of them must be declared upfront in `__init__`.

After you define your Module subclass, you can instantiate it as an object and call it just like the NN forward function in part II.

5.0.1 Module API: Two-Layer Network

Here is a concrete example of a 2-layer fully connected network:

[48]:
```
class TwoLayerFC(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super().__init__()
```

```
        # assign layer objects to class attributes
        self.fc1 = nn.Linear(input_size, hidden_size)
        # nn.init package contains convenient initialization methods
        # http://pytorch.org/docs/master/nn.html#torch-nn-init
        nn.init.kaiming_normal_(self.fc1.weight)
        self.fc2 = nn.Linear(hidden_size, num_classes)
        nn.init.kaiming_normal_(self.fc2.weight)

    def forward(self, x):
        # forward always defines connectivity
        x = flatten(x)
        scores = self.fc2(F.relu(self.fc1(x)))
        return scores

def test_TwoLayerFC():
    input_size = 50
    x = torch.zeros((64, input_size), dtype=dtype)  # minibatch size 64, feature␣
 ↪dimension 50
    model = TwoLayerFC(input_size, 42, 10)
    scores = model(x)
    print(scores.size())  # you should see [64, 10]
test_TwoLayerFC()
```

```
torch.Size([64, 10])
```

5.0.2 Module API: Three-Layer ConvNet

It's your turn to implement a 3-layer ConvNet followed by a fully connected layer. The network architecture should be the same as in Part II:

1. Convolutional layer with `channel_1` 5x5 filters with zero-padding of 2
2. ReLU
3. Convolutional layer with `channel_2` 3x3 filters with zero-padding of 1
4. ReLU
5. Fully-connected layer to `num_classes` classes

You should initialize the weight matrices of the model using the Kaiming normal initialization method.

HINT: http://pytorch.org/docs/stable/nn.html#conv2d

After you implement the three-layer ConvNet, the `test_ThreeLayerConvNet` function will run your implementation; it should print `(64, 10)` for the shape of the output scores.

[49]:
```
class ThreeLayerConvNet(nn.Module):
    def __init__(self, in_channel, channel_1, channel_2, num_classes):
        super().__init__()
        ########################################################################
        # TODO: Set up the layers you need for a three-layer ConvNet with the  #
```

```
        # architecture defined above.                                          #
        ################################################################################

        self.conv1 = nn.Conv2d(in_channel, channel_1, kernel_size=5, padding=2,␣
↪bias=True)
        nn.init.kaiming_normal_(self.conv1.weight)
        nn.init.constant_(self.conv1.bias, 0)

        self.conv2 = nn.Conv2d(channel_1, channel_2, kernel_size=3, padding=1,␣
↪bias=True)
        nn.init.kaiming_normal_(self.conv2.weight)
        nn.init.constant_(self.conv2.bias, 0)

        self.fc = nn.Linear(channel_2*32*32, num_classes)
        nn.init.kaiming_normal_(self.fc.weight)
        nn.init.constant_(self.fc.bias, 0)

        ################################################################################
        #                            END OF YOUR CODE                                  #␣
↪
        ################################################################################

    def forward(self, x):
        scores = None
        ################################################################################
        # TODO: Implement the forward function for a 3-layer ConvNet. you              #
        # should use the layers you defined in __init__ and specify the                #
        # connectivity of those layers in forward()                                    #
        ################################################################################

        relu1 = F.relu(self.conv1(x))
        relu2 = F.relu(self.conv2(relu1))
        scores = self.fc(flatten(relu2))

        ################################################################################
        #                                END OF YOUR CODE                              #
        ################################################################################
        return scores


def test_ThreeLayerConvNet():
    x = torch.zeros((64, 3, 32, 32), dtype=dtype)  # minibatch size 64, image␣
↪size [3, 32, 32]
    model = ThreeLayerConvNet(in_channel=3, channel_1=12, channel_2=8,␣
↪num_classes=10)
    scores = model(x)
    print(scores.size())  # you should see [64, 10]
```

```
test_ThreeLayerConvNet()
```

```
torch.Size([64, 10])
```

5.0.3 Module API: Check Accuracy

Given the validation or test set, we can check the classification accuracy of a neural network.

This version is slightly different from the one in part II. You don't manually pass in the parameters anymore.

[50]:
```
def check_accuracy_part34(loader, model):
    if loader.dataset.train:
        print('Checking accuracy on validation set')
    else:
        print('Checking accuracy on test set')
    num_correct = 0
    num_samples = 0
    model.eval()  # set model to evaluation mode
    with torch.no_grad():
        for x, y in loader:
            x = x.to(device=device, dtype=dtype)  # move to device, e.g. GPU
            y = y.to(device=device, dtype=torch.long)
            scores = model(x)
            _, preds = scores.max(1)
            num_correct += (preds == y).sum()
            num_samples += preds.size(0)
        acc = float(num_correct) / num_samples
        print('Got %d / %d correct (%.2f)' % (num_correct, num_samples, 100 *␣
↪acc))
```

5.0.4 Module API: Training Loop

We also use a slightly different training loop. Rather than updating the values of the weights ourselves, we use an Optimizer object from the `torch.optim` package, which abstract the notion of an optimization algorithm and provides implementations of most of the algorithms commonly used to optimize neural networks.

[51]:
```
def train_part34(model, optimizer, epochs=1):
    """
    Train a model on CIFAR-10 using the PyTorch Module API.

    Inputs:
    - model: A PyTorch Module giving the model to train.
    - optimizer: An Optimizer object we will use to train the model
    - epochs: (Optional) A Python integer giving the number of epochs to train␣
↪for
```

```
    Returns: Nothing, but prints model accuracies during training.
    """
    model = model.to(device=device)  # move the model parameters to CPU/GPU
    for e in range(epochs):
        for t, (x, y) in enumerate(loader_train):
            model.train()  # put model to training mode
            x = x.to(device=device, dtype=dtype)  # move to device, e.g. GPU
            y = y.to(device=device, dtype=torch.long)

            scores = model(x)
            loss = F.cross_entropy(scores, y)

            # Zero out all of the gradients for the variables which the optimizer
            # will update.
            optimizer.zero_grad()

            # This is the backwards pass: compute the gradient of the loss with
            # respect to each  parameter of the model.
            loss.backward()

            # Actually update the parameters of the model using the gradients
            # computed by the backwards pass.
            optimizer.step()

            if t % print_every == 0:
                print('Iteration %d, loss = %.4f' % (t, loss.item()))
                check_accuracy_part34(loader_val, model)
                print()
```

5.0.5 Module API: Train a Two-Layer Network

Now we are ready to run the training loop. In contrast to part II, we don't explicitly allocate parameter tensors anymore.

Simply pass the input size, hidden layer size, and number of classes (i.e. output size) to the constructor of `TwoLayerFC`.

You also need to define an optimizer that tracks all the learnable parameters inside `TwoLayerFC`.

You don't need to tune any hyperparameters, but you should see model accuracies above 40% after training for one epoch.

[52]:
```
hidden_layer_size = 4000
learning_rate = 1e-2
model = TwoLayerFC(3 * 32 * 32, hidden_layer_size, 10)
optimizer = optim.SGD(model.parameters(), lr=learning_rate)

train_part34(model, optimizer)
```

```
Iteration 0, loss = 3.9871
Checking accuracy on validation set
Got 149 / 1000 correct (14.90)

Iteration 100, loss = 2.5517
Checking accuracy on validation set
Got 307 / 1000 correct (30.70)

Iteration 200, loss = 2.3944
Checking accuracy on validation set
Got 367 / 1000 correct (36.70)

Iteration 300, loss = 1.7624
Checking accuracy on validation set
Got 390 / 1000 correct (39.00)

Iteration 400, loss = 1.9860
Checking accuracy on validation set
Got 432 / 1000 correct (43.20)

Iteration 500, loss = 2.0620
Checking accuracy on validation set
Got 396 / 1000 correct (39.60)

Iteration 600, loss = 1.9414
Checking accuracy on validation set
Got 426 / 1000 correct (42.60)

Iteration 700, loss = 2.0682
Checking accuracy on validation set
Got 429 / 1000 correct (42.90)
```

5.0.6 Module API: Train a Three-Layer ConvNet

You should now use the Module API to train a three-layer ConvNet on CIFAR. This should look very similar to training the two-layer network! You don't need to tune any hyperparameters, but you should achieve above above 45% after training for one epoch.

You should train the model using stochastic gradient descent without momentum.

[53]:
```
learning_rate = 3e-3
channel_1 = 32
channel_2 = 16

model = None
optimizer = None
################################################################################
```

```
# TODO: Instantiate your ThreeLayerConvNet model and a corresponding optimizer #
################################################################################

model = ThreeLayerConvNet(3, channel_1, channel_2, 10)
optimizer = optim.SGD(model.parameters(), lr=learning_rate)

################################################################################
#                                END OF YOUR CODE
################################################################################

train_part34(model, optimizer)
```

```
Iteration 0, loss = 3.0184
Checking accuracy on validation set
Got 166 / 1000 correct (16.60)

Iteration 100, loss = 1.8025
Checking accuracy on validation set
Got 342 / 1000 correct (34.20)

Iteration 200, loss = 1.9067
Checking accuracy on validation set
Got 383 / 1000 correct (38.30)

Iteration 300, loss = 1.4548
Checking accuracy on validation set
Got 420 / 1000 correct (42.00)

Iteration 400, loss = 1.4959
Checking accuracy on validation set
Got 440 / 1000 correct (44.00)

Iteration 500, loss = 1.5693
Checking accuracy on validation set
Got 478 / 1000 correct (47.80)

Iteration 600, loss = 1.4604
Checking accuracy on validation set
Got 496 / 1000 correct (49.60)

Iteration 700, loss = 1.5508
Checking accuracy on validation set
Got 488 / 1000 correct (48.80)
```

6 Part IV. PyTorch Sequential API

Part III introduced the PyTorch Module API, which allows you to define arbitrary learnable layers and their connectivity.

For simple models like a stack of feed forward layers, you still need to go through 3 steps: subclass `nn.Module`, assign layers to class attributes in `__init__`, and call each layer one by one in `forward()`. Is there a more convenient way?

Fortunately, PyTorch provides a container Module called `nn.Sequential`, which merges the above steps into one. It is not as flexible as `nn.Module`, because you cannot specify more complex topology than a feed-forward stack, but it's good enough for many use cases.

6.0.1 Sequential API: Two-Layer Network

Let's see how to rewrite our two-layer fully connected network example with `nn.Sequential`, and train it using the training loop defined above.

Again, you don't need to tune any hyperparameters here, but you shoud achieve above 40% accuracy after one epoch of training.

[54]:
```
# We need to wrap `flatten` function in a module in order to stack it
# in nn.Sequential
class Flatten(nn.Module):
    def forward(self, x):
        return flatten(x)

hidden_layer_size = 4000
learning_rate = 1e-2

model = nn.Sequential(
    Flatten(),
    nn.Linear(3 * 32 * 32, hidden_layer_size),
    nn.ReLU(),
    nn.Linear(hidden_layer_size, 10),
)

# you can use Nesterov momentum in optim.SGD
optimizer = optim.SGD(model.parameters(), lr=learning_rate,
                      momentum=0.9, nesterov=True)

train_part34(model, optimizer)
```

```
Iteration 0, loss = 2.4150
Checking accuracy on validation set
Got 138 / 1000 correct (13.80)

Iteration 100, loss = 2.0757
Checking accuracy on validation set
Got 386 / 1000 correct (38.60)
```

```
Iteration 200, loss = 1.6169
Checking accuracy on validation set
Got 397 / 1000 correct (39.70)

Iteration 300, loss = 1.8913
Checking accuracy on validation set
Got 430 / 1000 correct (43.00)

Iteration 400, loss = 1.9335
Checking accuracy on validation set
Got 393 / 1000 correct (39.30)

Iteration 500, loss = 1.8239
Checking accuracy on validation set
Got 456 / 1000 correct (45.60)

Iteration 600, loss = 1.6125
Checking accuracy on validation set
Got 423 / 1000 correct (42.30)

Iteration 700, loss = 1.8768
Checking accuracy on validation set
Got 426 / 1000 correct (42.60)
```

6.0.2 Sequential API: Three-Layer ConvNet

Here you should use `nn.Sequential` to define and train a three-layer ConvNet with the same architecture we used in Part III:

1. Convolutional layer (with bias) with 32 5x5 filters, with zero-padding of 2
2. ReLU
3. Convolutional layer (with bias) with 16 3x3 filters, with zero-padding of 1
4. ReLU
5. Fully-connected layer (with bias) to compute scores for 10 classes

You should initialize your weight matrices using the `random_weight` function defined above, and you should initialize your bias vectors using the `zero_weight` function above.

You should optimize your model using stochastic gradient descent with Nesterov momentum 0.9.

Again, you don't need to tune any hyperparameters but you should see accuracy above 55% after one epoch of training.

[55]:
```
channel_1 = 32
channel_2 = 16
learning_rate = 1e-2

model = None
```

```
optimizer = None

################################################################################
# TODO: Rewrite the 3-layer ConvNet with bias from Part III with the           #
# Sequential API.                                                              #
################################################################################

model = nn.Sequential(
    nn.Conv2d(3, channel_1, kernel_size=5, padding=2),
    nn.ReLU(),
    nn.Conv2d(channel_1, channel_2, kernel_size=3, padding=1),
    nn.ReLU(),
    Flatten(),
    nn.Linear(channel_2*32*32, 10),
)

optimizer = optim.SGD(model.parameters(), lr=learning_rate,
                      momentum=0.9, nesterov=True)

# Weight initialization
# Ref: http://pytorch.org/docs/stable/nn.html#torch.nn.Module.apply
def init_weights(m):
    # print(m)
    if type(m) == nn.Conv2d or type(m) == nn.Linear:
        random_weight(m.weight.size())
        zero_weight(m.bias.size())

model.apply(init_weights)

################################################################################
#                                 END OF YOUR CODE
################################################################################

train_part34(model, optimizer)
```

```
Iteration 0, loss = 2.3311
Checking accuracy on validation set
Got 104 / 1000 correct (10.40)

Iteration 100, loss = 1.3677
Checking accuracy on validation set
Got 450 / 1000 correct (45.00)

Iteration 200, loss = 1.6045
Checking accuracy on validation set
Got 485 / 1000 correct (48.50)
```

```
Iteration 300, loss = 1.3959
Checking accuracy on validation set
Got 533 / 1000 correct (53.30)

Iteration 400, loss = 1.4906
Checking accuracy on validation set
Got 525 / 1000 correct (52.50)

Iteration 500, loss = 1.3200
Checking accuracy on validation set
Got 542 / 1000 correct (54.20)

Iteration 600, loss = 1.2658
Checking accuracy on validation set
Got 578 / 1000 correct (57.80)

Iteration 700, loss = 1.1156
Checking accuracy on validation set
Got 576 / 1000 correct (57.60)
```

7 Part V. CIFAR-10 open-ended challenge

In this section, you can experiment with whatever ConvNet architecture you'd like on CIFAR-10.

Now it's your job to experiment with architectures, hyperparameters, loss functions, and optimizers to train a model that achieves **at least 70%** accuracy on the CIFAR-10 **validation** set within 10 epochs. You can use the check_accuracy and train functions from above. You can use either `nn.Module` or `nn.Sequential` API.

Describe what you did at the end of this notebook.

Here are the official API documentation for each component. One note: what we call in the class "spatial batch norm" is called "BatchNorm2D" in PyTorch.

- Layers in torch.nn package: http://pytorch.org/docs/stable/nn.html
- Activations: http://pytorch.org/docs/stable/nn.html#non-linear-activations
- Loss functions: http://pytorch.org/docs/stable/nn.html#loss-functions
- Optimizers: http://pytorch.org/docs/stable/optim.html

7.0.1 Things you might try:

- **Filter size**: Above we used 5x5; would smaller filters be more efficient?
- **Number of filters**: Above we used 32 filters. Do more or fewer do better?
- **Pooling vs Strided Convolution**: Do you use max pooling or just stride convolutions?
- **Batch normalization**: Try adding spatial batch normalization after convolution layers and vanilla batch normalization after affine layers. Do your networks train faster?
- **Network architecture**: The network above has two layers of trainable parameters. Can you do better with a deep network? Good architectures to try include:

  - [conv-relu-pool]xN -> [affine]xM -> [softmax or SVM]
  - [conv-relu-conv-relu-pool]xN -> [affine]xM -> [softmax or SVM]
  - [batchnorm-relu-conv]xN -> [affine]xM -> [softmax or SVM]
- **Global Average Pooling**: Instead of flattening and then having multiple affine layers, perform convolutions until your image gets small (7x7 or so) and then perform an average pooling operation to get to a 1x1 image picture (1, 1 , Filter#), which is then reshaped into a (Filter#) vector. This is used in Google's Inception Network (See Table 1 for their architecture).
- **Regularization**: Add l2 weight regularization, or perhaps use Dropout.

7.0.2 Tips for training

For each network architecture that you try, you should tune the learning rate and other hyperparameters. When doing this there are a couple important things to keep in mind:

- If the parameters are working well, you should see improvement within a few hundred iterations
- Remember the coarse-to-fine approach for hyperparameter tuning: start by testing a large range of hyperparameters for just a few training iterations to find the combinations of parameters that are working at all.
- Once you have found some sets of parameters that seem to work, search more finely around these parameters. You may need to train for more epochs.
- You should use the validation set for hyperparameter search, and save your test set for evaluating your architecture on the best parameters as selected by the validation set.

7.0.3 Going above and beyond

If you are feeling adventurous there are many other features you can implement to try and improve your performance. You are **not required** to implement any of these, but don't miss the fun if you have time!

- Alternative optimizers: you can try Adam, Adagrad, RMSprop, etc.
- Alternative activation functions such as leaky ReLU, parametric ReLU, ELU, or MaxOut.
- Model ensembles
- Data augmentation
- New Architectures
  - ResNets where the input from the previous layer is added to the output.
  - DenseNets where inputs into previous layers are concatenated together.
  - This blog has an in-depth overview

7.0.4 Have fun and happy training!

[58]:
```
################################################################################
# TODO:                                                                        #␣
 ↪
# Experiment with any architectures, optimizers, and hyperparameters.          #
# Achieve AT LEAST 70% accuracy on the *validation set* within 10 epochs.      #
#                                                                              #
# Note that you can use the check_accuracy function to evaluate on either      #
```

```
# the test set or the validation set, by passing either loader_test or         #
# loader_val as the second argument to check_accuracy. You should not touch    #
# the test set until you have finished your architecture and  hyperparameter   #
# tuning, and only run the test set once at the end to report a final value.   #
################################################################################
model = None
optimizer = None

# A 4-layer convolutional network
# (conv -> batchnorm -> relu -> maxpool) * 3 -> fc
layer1 = nn.Sequential(
    nn.Conv2d(3, 16, kernel_size=5, padding=2),
    nn.BatchNorm2d(16),
    nn.ReLU(),
    nn.MaxPool2d(2)
)

layer2 = nn.Sequential(
    nn.Conv2d(16, 32, kernel_size=3, padding=1),
    nn.BatchNorm2d(32),
    nn.ReLU(),
    nn.MaxPool2d(2)
)

layer3 = nn.Sequential(
    nn.Conv2d(32, 64, kernel_size=3, padding=1),
    nn.BatchNorm2d(64),
    nn.ReLU(),
    nn.MaxPool2d(2)
)

layer4 = nn.Sequential(
    nn.Dropout(0.3),
    nn.Linear(64*4*4, 10),
    nn.ReLU(),
    nn.Linear(10, 10),
)



model = nn.Sequential(
    layer1,
    layer2,
    layer3,
    Flatten(),
    layer4
)
```

```
learning_rate = 1.5e-3

optimizer = optim.Adam(model.parameters(), lr=learning_rate)

# Print training status every epoch: set print_every to a large number
print_every = 10000

################################################################################
#                                END OF YOUR CODE
################################################################################

# You should get at least 70% accuracy
train_part34(model, optimizer, epochs=10)
```

```
Iteration 0, loss = 2.3330
Checking accuracy on validation set
Got 131 / 1000 correct (13.10)

Iteration 0, loss = 1.2069
Checking accuracy on validation set
Got 562 / 1000 correct (56.20)

Iteration 0, loss = 1.0123
Checking accuracy on validation set
Got 636 / 1000 correct (63.60)

Iteration 0, loss = 1.1918
Checking accuracy on validation set
Got 683 / 1000 correct (68.30)

Iteration 0, loss = 0.8696
Checking accuracy on validation set
Got 690 / 1000 correct (69.00)

Iteration 0, loss = 0.5637
Checking accuracy on validation set
Got 678 / 1000 correct (67.80)

Iteration 0, loss = 0.8290
Checking accuracy on validation set
Got 712 / 1000 correct (71.20)

Iteration 0, loss = 0.6019
Checking accuracy on validation set
Got 730 / 1000 correct (73.00)
```

```
Iteration 0, loss = 0.8366
Checking accuracy on validation set
Got 728 / 1000 correct (72.80)

Iteration 0, loss = 0.7205
Checking accuracy on validation set
Got 741 / 1000 correct (74.10)
```

7.1 Describe what you did

In the cell below you should write an explanation of what you did, any additional features that you implemented, and/or any graphs that you made in the process of training and evaluating your network.

TODO: Describe what you did

I did some tests and managed to increase the test accuracy to 73.4%. To do this I did not touch the convolutional layers and simply extended the fully connected layers, named layer 4. I used a dropout layer after the convolutional layers to decrease the chance of overfitting with drop ratte 0.6. This also allowed me to use a larger learning rate since I dropped the chance of overfitting significantly. After the dropout layer, I used a ReLU layer to increase complex learning. Its not too much of an increase but it was great practice that will prove beneficial in the final project.

7.2 Test set – run this only once

Now that we've gotten a result we're happy with, we test our final model on the test set (which you should store in best_model). Think about how this compares to your validation set accuracy.

[59]:
```
best_model = model
check_accuracy_part34(loader_test, best_model)
```

```
Checking accuracy on test set
Got 7340 / 10000 correct (73.40)
```

[ ]:
```
```

6 Written Code for Questions 1 and 3

The following pages in this section contain the written code for Question 1 and 3

```
import h5py
import numpy as np
from matplotlib import pyplot as plt
import seaborn as sn
import time
import sys


########################################
#  QUESTION 1
########################################


class Q1AutoEncoder(object):
    """
    Autoencoder class for Question 1
    """


    def init_params(self, Lin, Lhid):
        """
        A function which initializes the weights following the assignment
         requirements
        @param Lin: The input layer size, 256
        @param Lhid: The hidden layer size
        @return: The initialized weights and their corresponding momentum
         values
        """

        Lout = Lin

        r = np.sqrt(6 / (Lin + Lhid))
        W1 = np.random.uniform(-r, r, size=(Lin, Lhid))
        b1 = np.random.uniform(-r, r, size=(1, Lhid))

        r = np.sqrt(6 / (Lhid + Lout))
        W2 = W1.T
        b2 = np.random.uniform(-r, r, size=(1, Lout))

        We = (W1, W2, b1, b2)
        mWe = (0, 0, 0, 0)

        return We, mWe


    def train(self, data, params, eta=0.1, alpha=0.9, epoch=10,
     batch_size=None):
        """
        The training function. Runs epochs and trains the given data. For this
         question this is
        used for the autoencoder.
```

```
@param data: the training data
@param params: the required parameters, given in the assignment
@param eta: learning ratet
@param alpha: momentum multiplier
@param epoch: the epoch number for training
@param batch_size: batch size for SGD
@return: the weights (exttracted features) and the loss
"""

J_list = []
if batch_size is None:
    batch_size = data.shape[0]

Lin = params["Lin"]
Lhid = params["Lhid"]
We, mWe = self.init_params(Lin,  Lhid)

iter_per_epoch = int(data.shape[0] / batch_size)

for i in range(epoch):

    time_start = time.time()

    J_total = 0

    start = 0
    end = batch_size

    p = np.random.permutation(data.shape[0])
    data = data[p]

    mWe = (0, 0, 0, 0)

    for j in range(iter_per_epoch):

        batchData = data[start:end]

        J, Jgrad, cache = self.aeCost(We, batchData, params)
        We, mWe = self.solver(Jgrad, cache, We, mWe, eta, alpha)

        J_total += J
        start = end
        end += batch_size

    time_remain = (epoch - i - 1) * (time.time() - time_start)
    if time_remain < 60:
        time_remain = round(time_remain)
        time_label = "second(s)"
    else:
        time_remain = round(time_remain / 60)
        time_label = "minute(s)"
```

```
            J_total = J_total/iter_per_epoch

            print("Loss: {:.2f} [Epoch {} of {}, ETA: {} {}]".format(J_total,
             i+1, epoch, time_remain, time_label))
            J_list.append(J_total)

        print("\n")

        return We, J_list


    def aeCost(self, We, data, params):
        """
        This function finds the first error gradients and does forward pass
        @param We: Weights
        @param data: training data, this comes in as the batch data
        @param params: the parameters
        @return: returns the error gradients and derivative and other
         variables via cahce
        """

        N = data.shape[0]

        W1, W2, b1, b2 = We

        rho = params["rho"]
        beta = params["beta"]
        lmb = params["lmb"]
        Lin = params["Lin"]
        Lhid = params["Lhid"]

        u = data @ W1 + b1
        h, h_drv = self.sigmoid(u)  # N x Lhid
        # h, h_drv = self.tanh(u)  # N x Lhid
        v = h @ W2 + b2
        o, o_drv = self.sigmoid(v)  # N x Lin
        # o, o_drv = self.tanh(v)  # N x Lin

        rho_b = h.mean(axis=0, keepdims=True)  # 1 x Lhid



        loss = 0.5/N * (np.linalg.norm(data - o, axis=1) ** 2).sum()
        tykhonov = 0.5 * lmb * (np.sum(W1 ** 2) + np.sum(W2 ** 2))
        KL = rho * np.log(rho/rho_b) + (1 - rho) * np.log((1 - rho)/(1 -
         rho_b))
        KL = beta * KL.sum()

        J = loss + tykhonov + KL
        #FIXME
```

```
    dloss = -(data - o)/N
    dtyk2 = lmb * W2
    dtyk1 = lmb * W1
    dKL = beta * (- rho/rho_b + (1-rho)/(1 - rho_b))/N

    cache = (data, h, h_drv, o_drv)
    Jgrad = (dloss, dtyk2, dtyk1, dKL)


    return J, Jgrad, cache


def solver(self, Jgrad, cache, We, mWe, eta, alpha):
    """
    Finds weight updates and updates them
    @param Jgrad: Error gradients
    @param cache: cache of variables coming from aeCost, needed for updates
    @param We: weights
    @param mWe: corresponding momentum terms
    @param eta: learning ratte
    @param alpha: momentum multiplier
    @return:
    """

    W1, W2, b1, b2 = We

    dW1 = 0
    dW2 = 0
    db1 = 0
    db2 = 0

    data, h, h_drv, o_drv = cache
    dloss, dtyk2, dtyk1, dKL = Jgrad

    delta = dloss * o_drv


    dW2 = h.T @ delta + dtyk2
    db2 = delta.sum(axis=0, keepdims=True)

    delta = h_drv * (delta @ W2.T + dKL)

    dW1 = data.T @ delta + dtyk1
    db1 = delta.sum(axis=0, keepdims=True)

    # FIXME
    dW2 = (dW1.T + dW2)/2
    dW1 = dW2.T

    dWe = (dW1, dW2, db1, db2)
```

```
        We, mWe = self.update(We, mWe, dWe, eta, alpha)

        return We, mWe


    def update(self, We, mWe, dWe, eta, alpha):
        """
        Updates weights
        @param We: weights
        @param mWe:momentum terms
        @param dWe: updates
        @param eta: learning rate
        @param alpha: mometum multiplier
        @return:updated weights and momentum terms
        """

        W1, W2, b1, b2 = We
        dW1, dW2, db1, db2 = dWe
        mW1, mW2, mb1, mb2 = mWe

        mW1 = eta * dW1 + alpha * mW1
        mW2 = eta * dW2 + alpha * mW2
        mb1 = eta * db1 + alpha * mb1
        mb2 = eta * db2 + alpha * mb2

        W1 -= mW1
        W2 -= mW2
        b1 -= mb1
        b2 -= mb2
        assert (W1 == W2.T).all()
        We = (W1, W2, b1, b2)
        mWe = (mW1, mW2, mb1, mb2)

        return We, mWe


    def predict(self, data, We):
        """
        Predicts the output, aka does forward pass
        @param data: input data
        @param We: weights
        @return: output
        """

        W1, W2, b1, b2 = We

        u = data @ W1 + b1
        h = self.sigmoid(u)[0]
        v = h @ W2 + b2
        o = self.sigmoid(v)[0]
        return o
```

```
    def sigmoid(self, X):
        """
        Sigmoid function
        @param X: input
        @return: output and derivative
        """
        a = 1 / (1 + np.exp(-X))
        d = a * (1 - a)
        return a, d


def normalize(X):
    """
    Normalizes given input
    @param X: input
    @return: normalized X
    """

    return (X - X.min())/(X.max() - X.min())


def plot(W, name, dim1, dim2):
    """
    A function which plots the weights for Q1
    @param W: Weight
    @param name: filename
    @param dim1: width
    @param dim2: height
    """

    fig, ax = plt.subplots(dim2, dim1, figsize=(dim1, dim2), dpi=320,
     facecolor='w', edgecolor='k')
    k = 0
    for i in range(dim2):
        for j in range(dim1):
            ax[i, j].imshow(W[k], cmap='gray')
            ax[i, j].axis("off")
            k += 1

    fig.subplots_adjust(wspace=0.1, hspace=0.1, left=0, right=1, bottom=0,
     top=1)
    fig.savefig(name + ".png")
    plt.close(fig)


def q1():
    filename = "assign3_data1.h5"
    h5 = h5py.File(filename, 'r')
    data = h5['data'][()].astype('float64')
```

```
h5.close()

# convert to grayscale using the luminosity model
data_n = 0.2126 * data[:, 0] + 0.7152 * data[:, 1] + 0.0722 * data[:, 2]

# normalize
assert data_n.shape[1] == data_n.shape[2]
dim = data_n.shape[1]
data_n = np.reshape(data_n, (data_n.shape[0], dim ** 2))  # flatten

data_n = data_n - data_n.mean(axis=1, keepdims=True)  # differentiate per
 image
std = np.std(data_n)  # find std
data_n = np.clip(data_n, - 3 * std, 3 * std)  # clip -+3 std
data_n = normalize(data_n)  # normalize to 0 - 1

data_n = 0.1 + data_n * 0.8  # map to 0.1 - 0.9
trainData = data_n

# plot 200 random images
data_n = np.reshape(data_n, (data_n.shape[0], dim, dim))  # reshape for
 imshow
data = data.transpose((0, 2, 3, 1))
fig1, ax1 = plt.subplots(10, 20, figsize=(20, 10))
fig2, ax2 = plt.subplots(10, 20, figsize=(20, 10), dpi=200, facecolor='w',
 edgecolor='k')

for i in range(10):
    for j in range(20):
        k = np.random.randint(0, data.shape[0])

        ax1[i, j].imshow(data[k].astype('float'))
        ax1[i, j].axis("off")

        ax2[i, j].imshow(data_n[k], cmap='gray')
        ax2[i, j].axis("off")

fig1.subplots_adjust(wspace=0, hspace=0, left=0, right=1, bottom=0, top=1)
fig2.subplots_adjust(wspace=0, hspace=0, left=0, right=1, bottom=0, top=1)
fig1.savefig("q1a_rgb.png")
fig2.savefig("q1a_gray.2.png")
plt.close("all")

eta = 0.075
alpha = 0.85
epoch = 200
batch_size = 32
rho = 0.025
beta = 2
lmb = 5e-4
Lin = trainData.shape[1]
```

```
    Lhid = 64

    params = {"rho": rho, "beta": beta, "lmb": lmb, "Lin": Lin, "Lhid": Lhid}
    ae = Q1AutoEncoder()
    w = ae.train(trainData, params, eta, alpha, epoch, batch_size)[0]
    W = normalize(w[0]).T
    W = W.reshape((W.shape[0], dim, dim))

    name =
     "rho={:.2f}|beta={:.2f}|eta={:.2f}|alpha={:
     .2f}|lambda={}|batch={}|Lhid={}".format(rho, beta, eta, alpha, lmb,
     batch_size, Lhid)
    wdim = int(np.sqrt(W.shape[0]))
    plot(W, name + "weights", wdim, wdim)


########################################
#  QUESTION 3
########################################


class Q3NET(object):
    """
    Network class for Q3. Implements RNN, LSTM and GRU.
    """


    def __init__(self, size, qPart):

        self.qPart = qPart
        self.size = size
        self.layer_size = len(size) - 1
        self.mlp_layer_size = None
        self.mlp_params, self.first_layer_params, self.mlp_momentum,
         self.first_layer_momentum = None, None, None, None
        self.init_params()


    def init_params(self):
        """
        Initializes parameters for MLP layers and also thte first layer.
        Uses Xavier Distribution for initialization.
        """


        qPart = self.qPart
        size = self.size
        layer_size = self.layer_size

        W = []
        b = []
```

```
for i in range(1, layer_size):
    # Xavier Uniform
    r = np.sqrt(6 / (size[i] + size[i + 1]))
    W.append(np.random.uniform(-r, r, size=(size[i], size[i + 1])))
    b.append(np.zeros((1, size[i + 1])))

self.mlp_layer_size = len(W)
params = {"W": W, "b": b}
momentum = {"W": [0] * self.mlp_layer_size, "b": [0] *
 self.mlp_layer_size}
self.mlp_params = params
self.mlp_momentum = momentum

N = size[0]
H = size[1]
Z = N + H

if qPart == 1:
    r = np.sqrt(6 / (N + H))
    Wih = np.random.uniform(-r, r, size=(N, H))
    r = np.sqrt(6 / (H + H))
    Whh = np.random.uniform(-r, r, size=(H, H))
    b = np.zeros((1, H))

    params = {"Wih": Wih, "Whh": Whh, "b": b}

if qPart == 2:
    r = np.sqrt(6 / (Z + H))

    Wf = np.random.uniform(-r, r, size=(Z, H))
    Wi = np.random.uniform(-r, r, size=(Z, H))
    Wc = np.random.uniform(-r, r, size=(Z, H))
    Wo = np.random.uniform(-r, r, size=(Z, H))

    bf = np.zeros((1, H))
    bi = np.zeros((1, H))
    bc = np.zeros((1, H))
    bo = np.zeros((1, H))

    params = {"Wf": Wf, "bf": bf,
              "Wi": Wi, "bi": bi,
              "Wc": Wc, "bc": bc,
              "Wo": Wo, "bo": bo}

if qPart == 3:
    rN = np.sqrt(6 / (N + H))
    rH = np.sqrt(6 / (H + H))

    Wz = np.random.uniform(-rN, rN, size=(N, H))
    Uz = np.random.uniform(-rH, rH, size=(H, H))
    bz = np.zeros((1, H))
```

```
        Wr = np.random.uniform(-rN, rN, size=(N, H))
        Ur = np.random.uniform(-rH, rH, size=(H, H))
        br = np.zeros((1, H))

        Wh = np.random.uniform(-rN, rN, size=(N, H))
        Uh = np.random.uniform(-rH, rH, size=(H, H))
        bh = np.zeros((1, H))

        params = {"Wz": Wz, "Uz": Uz, "bz": bz,
                  "Wr": Wr, "Ur": Ur, "br": br,
                  "Wh": Wh, "Uh": Uh, "bh": bh}


    momentum = dict.fromkeys(params.keys(), 0)
    self.first_layer_params = params
    self.first_layer_momentum = momentum


def train(self, X, Y, eta, alpha, batch_size, epoch):
    """
    Training function. Calls forward and backward pass. Uses SGD and
     trains with mini-batch. Ya'know, the regular stuff.
    @param X: Training data
    @param Y: Training labels
    @param eta: learning rate
    @param alpha: momentum multiplier
    @param batch_size: self-explanatory
    @param epoch: self-explanatory
    @return Loss Metrics
    """

    train_loss_list = []
    val_loss_list = []
    train_acc_list = []
    val_acc_list = []

    # create validation set
    val_size = int(X.shape[0] / 10)
    p = np.random.permutation(X.shape[0])
    valX = X[p][:val_size]
    valY = Y[p][:val_size]
    X = X[p][val_size:]
    Y = Y[p][val_size:]

    sample_size = X.shape[0]
    iter_per_epoch = int(sample_size / batch_size)

    for i in range(epoch):

        time_start = time.time()
```

```
start = 0
end = batch_size
p = np.random.permutation(X.shape[0])
X = X[p]
Y = Y[p]

for j in range(iter_per_epoch):

    batchX = X[start:end]
    batchY = Y[start:end]

    # forward
    pred, o, drv, h, h_drv, cache = self.forward_pass(batchX)

    # error gradient at last layer
    delta = pred
    delta[batchY == 1] -= 1
    delta = delta / batch_size

    # backward
    fl_grads, mlp_grads = self.backward_pass(batchX, o, drv,
     delta, h, h_drv, cache)

    # update
    self.update_params(eta, alpha, fl_grads, mlp_grads)

    start = end
    end += batch_size

# epoch end

# train loss
pred = self.predict(X, acc=False)
train_loss = self.cross_entropy(Y, pred)

# train acc
train_acc = self.predict(X, Y, acc=True)

# val acc
val_acc = self.predict(valX, valY, acc=True)

# val loss
pred = self.predict(valX, acc=False)
val_loss = self.cross_entropy(valY, pred)

# create time remaining
time_remain = (epoch - i - 1) * (time.time() - time_start)

if time_remain < 60:
    time_remain = round(time_remain)
```

```
            time_label = "second(s)"
        else:
            time_remain = round(time_remain/60)
            time_label = "minute(s)"

        print('Train Loss: %.2f, Val Loss: %.2f, Train Acc: %.2f, Val Acc:
         %.2f [Epoch: %d of %d, ETA: %d %s]'
              % (train_loss, val_loss, train_acc, val_acc, i + 1, epoch,
               time_remain, time_label))

        train_loss_list.append(train_loss)
        val_loss_list.append(val_loss)
        train_acc_list.append(train_acc)
        val_acc_list.append(val_acc)

        if i > 26:
            conv = val_loss_list[-16:-1]
            conv = sum(conv) / len(conv)

            limit = 0.02
            if (conv - limit) < val_loss < (conv + limit):
                print("\nTraining stopped since validation C-E reached
                 convergence.")
                return {"train_loss_list": train_loss_list,
                 "val_loss_list": val_loss_list,
                        "train_acc_list": train_acc_list, "val_acc_list":
                         val_acc_list}

    return {"train_loss_list": train_loss_list, "val_loss_list":
     val_loss_list,
            "train_acc_list": train_acc_list, "val_acc_list": val_acc_list}


def forward_pass(self, X):
    """
    Forward pass.
    @param X: Input data
    @return: Stuff needed to update the weights, such as derivatives and
     activations through the forward pass.
    """

    qPart = self.qPart
    mlp_p = self.mlp_params
    fl_p = self.first_layer_params

    o = []
    drv = []

    h = 0
    h_drv = 0
    cache = 0
```

```
    # first layer
    if qPart == 1:
        h, h_drv = self.forward_recurrent(X, fl_p)
        o.append(h[:, -1, :])
        drv.append(h_drv[:, -1, :])
    if qPart == 2:
        h, cache = self.forward_lstm(X, fl_p)
        o.append(h)
        drv.append(1)
    if qPart == 3:
        h, cache = self.forward_gru(X, fl_p)
        o.append(h)
        drv.append(1)

    # relu layers
    for i in range(self.mlp_layer_size - 1):
        activation, derivative = self.forward_perceptron(o[-1],
         mlp_p["W"][i], mlp_p["b"][i], "relu")
        o.append(activation)
        drv.append(derivative)

    # output layer
    pred = self.forward_perceptron(o[-1], mlp_p["W"][-1], mlp_p["b"][-1],
     "softmax")[0]

    return pred, o, drv, h, h_drv, cache


def backward_pass(self, X, o, drv, delta, h=None, h_drv=None, cache=None):
    """
    The backward pass function which calls network layers to obtain the
     gradients.
    @param X: training data
    @param o: activations of mlp layers
    @param drv: derivatives of mlp layers
    @param delta: The first error gradient for the backward pass, this
     getts updated through the layers and time
    @param h: only required for recurrent first layer, the activations for
     all time samples
    @param h_drv: only required for recurrent first layer, the derivatives
     of activations for all time samples
    @param cache: needed parameters to find the first layer updates
    @return: first layer gradients, mlp gradients
    """
    qPart = self.qPart
    fl_p = self.first_layer_params
    mlp_p = self.mlp_params

    fl_grads = dict.fromkeys(fl_p.keys())
```

```
    mlp_grads = {"W": [0] * self.mlp_layer_size, "b": [0] *
     self.mlp_layer_size}

    # backpropagation until recurrent
    for i in reversed(range(self.mlp_layer_size)):
        mlp_grads["W"][i], mlp_grads["b"][i], delta =
         self.backward_perceptron(mlp_p["W"][i], o[i], drv[i], delta)

    # backpropagation through time
    if qPart == 1:
        fl_grads = self.backward_recurrent(X, h, h_drv, delta, fl_p)
    if qPart == 2:
        fl_grads = self.backward_lstm(cache, fl_p, delta)
    if qPart == 3:
        fl_grads = self.backward_gru(X, cache, fl_p, delta)

    return fl_grads, mlp_grads


def update_params(self, eta, alpha, fl_grads, mlp_grads):
    """
    Updates parameters for first and mlp layers.
    @param eta: learning rate
    @param alpha: momentum multiplier
    @param fl_grads: first layer gradients
    @param mlp_grads: mlp gradients
    """

    # obtain
    fl_p = self.first_layer_params
    fl_m = self.first_layer_momentum
    mlp_p = self.mlp_params
    mlp_m = self.mlp_momentum

    # first layer
    for p in self.first_layer_params:
        fl_m[p] = eta * fl_grads[p] + alpha * fl_m[p]
        fl_p[p] -= fl_m[p]

    # mlp layers
    for i in range(self.mlp_layer_size):
        mlp_m["W"][i] = eta * mlp_grads["W"][i] + alpha * mlp_m["W"][i]
        mlp_m["b"][i] = eta * mlp_grads["b"][i] + alpha * mlp_m["b"][i]
        mlp_p["W"][i] -= mlp_m["W"][i]
        mlp_p["b"][i] -= mlp_m["b"][i]

    # update
    self.first_layer_params = fl_p
    self.first_layer_momentum = fl_m
    self.mlp_params = mlp_p
    self.mlp_momentum = mlp_m
```

```
def forward_perceptron(self, X, W, b, a):
    """
    Finds the activation and derivative for MLP layers
    @param X: input data
    @param W: weight
    @param b: bias
    @param a: the fundtion type (relu, sigmoid, tanh, softmax)
    @return: the activation and its derivative
    """
    u = X @ W + b
    return self.activation(u, a)


def backward_perceptron(self, W, o, drv, delta):
    """
    Finds the gradients of MLP layers
    @param W: weight
    @param o: past output, input of this layer
    @param drv: past output derivative, derivative of the input to this
     layer
    @param delta: the error gradient from the previous layer
    @return: gradients and the updated delta term
    """
    dW = o.T @ delta
    db = delta.sum(axis=0, keepdims=True)
    delta = drv * (delta @ W.T)
    return dW, db, delta


def forward_recurrent(self, X, fl_p):
    """
    Forward pass for the recurrent layer. Not very different from MLP
     except for the fact
    that its done over 150 time samples.
    @param X: input data
    @param fl_p: first layer parameters
    @return: the activations and their derivatives needed for backward pass
    """

    N, T, D = X.shape
    H = self.size[1]

    Wih = fl_p["Wih"]
    Whh = fl_p["Whh"]
    b = fl_p["b"]

    h_prev = np.zeros((N, H))
    h = np.empty((N, T, H))
    h_drv = np.empty((N, T, H))
```

```
    for t in range(T):
        x = X[:, t, :]
        u = x @ Wih + h_prev @ Whh + b
        h[:, t, :], h_drv[:, t, :] = self.activation(u, "tanh")
        h_prev = h[:, t, :]

    return h, h_drv


def backward_recurrent(self, X, h, h_drv, delta, fl_p):
    """
    Backwards pass for recurrent layer. This is very similar to the MLP
     backward propagation
    as it can be seen and the way delta is updated is also the same. BPTT
     is done to find the
    gradients, this means the gradients are summed up through 150 time
     steps.
    @param X: input data
    @param h: the 150 time sampled activations
    @param h_drv: their derivatives
    @param delta: the error gradient coming from the previous layer
    @param fl_p: first layer parameters
    @return: gradients
    """

    N, T, D = X.shape
    H = self.size[1]

    Whh = fl_p["Whh"]

    dWih = 0
    dWhh = 0
    db = 0

    for t in reversed(range(T)):
        x = X[:, t, :]

        if t > 0:
            h_prev = h[:, t - 1, :]
            h_prev_derv = h_drv[:, t - 1, :]
        else:
            h_prev = np.zeros((N, H))
            h_prev_derv = 0

        dWih += x.T @ delta
        dWhh += h_prev.T @ delta
        db += delta.sum(axis=0, keepdims=True)
        delta = h_prev_derv * (delta @ Whh)

    return {"Wih": dWih, "Whh": dWhh, "b": db}
```

```
def forward_lstm(self, X, fl_p):
    """
    Forward pass of LSTM. It might look a bit confusing but its not so
     different from the
    mathematical equations of the gates.
    @param X: input data
    @param fl_p: first layer parameters
    @return: the final h value (this is needed for th error gradient
     calculations
    of the first MLP layer) and cache which conttains needed variables for
     this layers backward pass.
    """

    N, T, D = X.shape
    H = self.size[1]

    Wf, bf = fl_p["Wf"], fl_p["bf"]
    Wi, bi = fl_p["Wi"], fl_p["bi"]
    Wc, bc = fl_p["Wc"], fl_p["bc"]
    Wo, bo = fl_p["Wo"], fl_p["bo"]

    h_prev = np.zeros((N, H))
    c_prev = np.zeros((N, H))
    z = np.empty((N, T, D + H))
    c = np.empty((N, T, H))
    tanhc = np.empty((N, T, H))
    hf = 0
    hi = np.empty((N, T, H))
    hc = np.empty((N, T, H))
    ho = np.empty((N, T, H))
    tanhc_d = np.empty((N, T, H))
    hf_d = np.empty((N, T, H))
    hi_d = np.empty((N, T, H))
    hc_d = np.empty((N, T, H))
    ho_d = np.empty((N, T, H))

    for t in range(T):
        z[:, t, :] = np.column_stack((h_prev, X[:, t, :]))
        z_cur = z[:, t, :]

        hf, hf_d[:, t, :] = self.activation(z_cur @ Wf + bf, "sigmoid")
        hi[:, t, :], hi_d[:, t, :] = self.activation(z_cur @ Wi + bi,
         "sigmoid")
        hc[:, t, :], hc_d[:, t, :] = self.activation(z_cur @ Wc + bc,
         "tanh")
        ho[:, t, :], ho_d[:, t, :] = self.activation(z_cur @ Wo + bo,
         "sigmoid")

        c[:, t, :] = hf * c_prev + hi[:, t, :] * hc[:, t, :]
```

```
            tanhc[:, t, :], tanhc_d[:, t, :] = self.activation(c[:, t, :],
             "tanh")
            h_prev = ho[:, t, :] * tanhc[:, t, :]
            c_prev = c[:, t, :]

            cache = {"z": z,
                     "c": c,
                     "tanhc": (tanhc, tanhc_d),
                     "hf_d": hf_d,
                     "hi": (hi, hi_d),
                     "hc": (hc, hc_d),
                     "ho": (ho, ho_d)}

        return h_prev, cache


    def backward_lstm(self, cache, fl_p, delta):
        """
        Backward propagation for LSTM.
        @param cache: has the needed variables for gradients
        @param fl_p: first layer parameters
        @param delta: error gradient from upper layer
        @return: gradients
        """
        # unpack variables
        Wf = fl_p["Wf"]
        Wi = fl_p["Wi"]
        Wc = fl_p["Wc"]
        Wo = fl_p["Wo"]

        z = cache["z"]
        c = cache["c"]
        tanhc, tanhc_d = cache["tanhc"]
        hf_d = cache["hf_d"]
        hi, hi_d = cache["hi"]
        hc, hc_d = cache["hc"]
        ho, ho_d = cache["ho"]

        H = self.size[1]
        T = z.shape[1]

        # initialize gradients to zero
        dWf = 0
        dWi = 0
        dWc = 0
        dWo = 0
        dbf = 0
        dbi = 0
        dbc = 0
        dbo = 0
```

```
    # BPTT starts
    for t in reversed(range(T)):

        z_cur = z[:, t, :]

        # if t = 0, c = 0
        if t > 0:
            c_prev = c[:, t - 1, :]
        else:
            c_prev = 0

        # firs find all 4 "gate gradients"
        # finding these first reduces clutter.
        dc = delta * ho[:, t, :] * tanhc_d[:, t, :]
        dhf = dc * c_prev * hf_d[:, t, :]
        dhi = dc * hc[:, t, :] * hi_d[:, t, :]
        dhc = dc * hi[:, t, :] * hc_d[:, t, :]
        dho = delta * tanhc[:, t, :] * ho_d[:, t, :]

        # add to all weights their respective values at that time
        dWf += z_cur.T @ dhf
        dbf += dhf.sum(axis=0, keepdims=True)

        dWi += z_cur.T @ dhi
        dbi += dhi.sum(axis=0, keepdims=True)

        dWc += z_cur.T @ dhc
        dbc += dhc.sum(axis=0, keepdims=True)

        dWo += z_cur.T @ dho
        dbo += dho.sum(axis=0, keepdims=True)

        # update the error gradient.
        # since weights are multiplied with a sttacked version of x, h(t-1)
        # we take only he weights of the previous layer by [:, :H]
        dxf = dhf @ Wf.T[:, :H]
        dxi = dhi @ Wi.T[:, :H]
        dxc = dhc @ Wc.T[:, :H]
        dxo = dho @ Wo.T[:, :H]

        delta = (dxf + dxi + dxc + dxo)  # we add them up

    grads = {"Wf": dWf, "bf": dbf,
             "Wi": dWi, "bi": dbi,
             "Wc": dWc, "bc": dbc,
             "Wo": dWo, "bo": dbo}

    return grads


def forward_gru(self, X, fl_p):
```

```
"""
Forward pass for GRU. Again, tthe same as the respective mathematical
 equations listed out for GRU.
@param X: input data
@param fl_p: first layer parameters.
@return: the final activation value and cache for backprop
"""

Wz = fl_p["Wz"]
Wr = fl_p["Wr"]
Wh = fl_p["Wh"]

Uz = fl_p["Uz"]
Ur = fl_p["Ur"]
Uh = fl_p["Uh"]

bz = fl_p["bz"]
br = fl_p["br"]
bh = fl_p["bh"]

N, T, D = X.shape
H = self.size[1]

h_prev = np.zeros((N, H))

z = np.empty((N, T, H))
z_d = np.empty((N, T, H))
r = np.empty((N, T, H))
r_d = np.empty((N, T, H))
h_tilde = np.empty((N, T, H))
h_tilde_d = np.empty((N, T, H))
h = np.empty((N, T, H))

for t in range(T):
    x = X[:, t, :]
    z[:, t, :], z_d[:, t, :] = self.activation(x @ Wz + h_prev @ Uz +
     bz, "sigmoid")
    r[:, t, :], r_d[:, t, :] = self.activation(x @ Wr + h_prev @ Ur +
     br, "sigmoid")
    h_tilde[:, t, :], h_tilde_d[:, t, :] = self.activation(x @ Wh +
     (r[:, t, :] * h_prev) @ Uh + bh, "tanh")
    h[:, t, :] = (1 - z[:, t, :]) * h_prev + z[:, t, :] * h_tilde[:,
     t, :]

    h_prev = h[:, t, :]

cache = {"z": (z, z_d),
         "r": (r, r_d),
         "h_tilde": (h_tilde, h_tilde_d),
         "h": h}
```

```
    return h_prev, cache


def backward_gru(self, X, cache, fl_p, delta):
    """
    Backpropagation for GRU. Much easier than LSTM although the errror
     gradient updates di get a bit confusing.
    I did all of these by hand btw, using the chain rule and the
     derivative of multiplications.
    @param X: input data
    @param cache: variables needed for backprop from the forward pass
    @param fl_p: first layer parameters
    @param delta: error gradient from upper layer
    @return: gradients
    """

    # unpack
    Uz = fl_p["Uz"]
    Ur = fl_p["Ur"]
    Uh = fl_p["Uh"]

    z, z_d = cache["z"]
    r, r_d = cache["r"]
    h_tilde, h_tilde_d = cache["h_tilde"]
    h = cache["h"]

    H = self.size[1]
    N, T, D = X.shape

    # initialize to zero since we are doing BPTT
    dWz = 0
    dUz = 0
    dbz = 0
    dWr = 0
    dUr = 0
    dbr = 0
    dWh = 0
    dUh = 0
    dbh = 0

    for t in reversed(range(T)):
        x = X[:, t, :]

        # if t = 0 we want h(t-1) = 0
        if t > 0:
            h_prev = h[:, t - 1, :]
        else:
            h_prev = np.zeros((N, H))

        # similar to LSTM we find some intermediate values for each gate
        # dE/dz is named as dz for example, this is true for all naming
```

```
            dz = delta * (h_tilde[:, t, :] - h_prev) * z_d[:, t, :]
            dh_tilde = delta * z[:, t, :] * h_tilde_d[:, t, :]
            dr = (dh_tilde @ Uh.T) * h_prev * r_d[:, t, :]

            # add to the sum of gradients
            dWz += x.T @ dz
            dUz += h_prev.T @ dz
            dbz += dz.sum(axis=0, keepdims=True)

            dWr += x.T @ dr
            dUr += h_prev.T @ dr
            dbr += dr.sum(axis=0, keepdims=True)

            dWh += x.T @ dh_tilde
            dUh += h_prev.T @ dh_tilde
            dbh += dh_tilde.sum(axis=0, keepdims=True)

            # update delta, this step uses chain rule and derivative of
             multiplication, at the end it simplifies to
            #the sum of these three terms
            d1 = delta * (1 - z[:, t, :])
            d2 = dz @ Uz.T
            d3 = (dh_tilde  @ Uh.T) * (r[:, t, :] + h_prev * (r_d[:, t, :] @
             Ur.T))

            delta = d1 + d2 + d3


        grads = {"Wz": dWz, "Uz": dUz, "bz": dbz,
                 "Wr": dWr, "Ur": dUr, "br": dbr,
                 "Wh": dWh, "Uh": dUh, "bh": dbh}

        return grads


    def cross_entropy(self, desired, output):
        """
        Cross entropy error
        @param desired: labels
        @param output: predictions
        @return: cross entropy error
        """
        return np.sum(- desired * np.log(output)) / desired.shape[0]


    def activation(self, X, a):
        """
        Function which outputs activation and derivative calculations
        @param X: input data
        @param a: activation type
        @return: activation, its derivative
```

```
    """

    if a == "tanh":
        activation = np.tanh(X)
        derivative = 1 - activation ** 2
        return activation, derivative

    if a == "sigmoid":
        activation = 1 / (1 + np.exp(-X))
        derivative = activation * (1 - activation)
        return activation, derivative

    if a == "relu":
        activation = X * (X > 0)
        derivative = 1 * (X > 0)
        return activation, derivative

    if a == "softmax":
        activation = np.exp(X) / np.sum(np.exp(X), axis=1, keepdims=True)
        derivative = None
        return activation, derivative


def predict(self, X, Y=None, acc=True, confusion = False):
    """
    The predict function, which does forward pass and then either returns
     the raw prediction or with labels
    returns the accuracy, or can also create the confusion matrix.
    @param X: Input matrix, these are not labels just data
    @param Y: Labels, ground truth, actual values
    @param acc: If true, computes the argmax version of prediction
    @param confusion: If true, gives the confusion matrix
    @return: prediction, accuracy or confusion matrix
    """
    pred = self.forward_pass(X)[0]

    if not acc:
        return pred

    pred = pred.argmax(axis=1)
    Y = Y.argmax(axis=1)

    if not confusion:
        return (pred == Y).mean() * 100 #accuracy

    K = len(np.unique(Y))  # Number of classes
    c = np.zeros((K, K))

    for i in range(len(Y)):
        c[Y[i]][pred[i]] += 1
```

```
        return c


def q3():
    filename = "assign3_data3.h5"
    h5 = h5py.File(filename, 'r')
    trX = h5['trX'][()].astype('float64')
    tstX = h5['tstX'][()].astype('float64')
    trY = h5['trY'][()].astype('float64')
    tstY = h5['tstY'][()].astype('float64')
    h5.close()

    print("\n!!DISCLAIMER!!\nThe code runs slow in the training stage, hence
     the epoch number has ben selected as 10 for all layers.\n"
          "The plotted graphs in the report show the whole 50 epoch run. If
           you desire to see the whole run, just change the epoch variable to
           50.\n\n")

    alpha = 0.85
    eta = 0.01
    epoch = 10
    batch_size = 32
    size = [trX.shape[2], 128, 32, 16, 6]

    print("Recurrent Layer\n")
    nn = Q3NET(size, 1)
    train_loss_list, val_loss_list, train_acc_list, val_acc_list =
     nn.train(trX, trY, eta, alpha, batch_size, epoch).values()
    tst_acc = nn.predict(tstX, tstY, acc=True)

    print("\nTest Accuracy: ", tst_acc, "\n\n")

    fig = plt.figure(figsize=(20, 10), dpi=160, facecolor='w', edgecolor='k')
    fig.suptitle("RNN\nLearning Rate {} | Momentum = {} | Batch Size  = {} |
     Hidden Layers = {}\n"
                 "Train Accuracy: {:.1f} | Validation Accuracy: {:.1f} | Test
                  Accuracy: {:.1f}\n "
                 .format(eta, alpha, batch_size, size[2:-1],
                  train_acc_list[-1], val_acc_list[-1], tst_acc), fontsize=13)

    plt.subplot(2, 2, 1)
    plt.plot(train_loss_list, "C2", label="Train Cross Entropy Loss")
    plt.title("Train Cross Entropy Loss")
    plt.xlabel("Epoch")
    plt.ylabel("Loss")
    plt.subplot(2, 2, 2)
    plt.plot(val_loss_list, "C3", label="Validation Cross Entropy Loss")
    plt.title("Validation Cross Entropy Loss")
    plt.xlabel("Epoch")
    plt.ylabel("Loss")
    plt.subplot(2, 2, 3)
```

```
plt.plot(train_acc_list, "C2", label="Train Accuracy")
plt.title("Train Accuracy")
plt.xlabel("Epoch")
plt.ylabel("Accuracy")
plt.subplot(2, 2, 4)
plt.plot(val_acc_list, "C3", label="Validation Accuracy")
plt.title("Validation Accuracy")
plt.xlabel("Epoch")
plt.ylabel("Accuracy")

plt.savefig("q3a.png", bbox_inches='tight')

train_confusion = nn.predict(trX, trY, acc=True, confusion=True)
test_confusion = nn.predict(tstX, tstY, acc=True, confusion=True)

plt.figure(figsize=(20, 10), dpi=160)

names = [1, 2, 3, 4, 5, 6]

plt.subplot(1, 2, 1)
sn.heatmap(train_confusion, annot=True, annot_kws={"size": 8},
 xticklabels=names, yticklabels=names, cmap=sn.cm.rocket_r, fmt='g')
plt.title("Train Confusion Matrix")
plt.ylabel("Actual")
plt.xlabel("Prediction")
plt.subplot(1, 2, 2)
sn.heatmap(test_confusion, annot=True, annot_kws={"size": 8},
 xticklabels=names, yticklabels=names, cmap=sn.cm.rocket_r, fmt='g')
plt.title("Test Confusion Matrix")
plt.ylabel("Actual")
plt.xlabel("Prediction")
plt.savefig("q3a_confusion.png", bbox_inches='tight')

##############################

alpha = 0.85
eta = 0.01
epoch = 10
batch_size = 32
size = [trX.shape[2], 128, 32, 16, 6]

print("\nLSTM Layer\n")

nn = Q3NET(size, 2)
train_loss_list, val_loss_list, train_acc_list, val_acc_list =
 nn.train(trX, trY, eta, alpha, batch_size, epoch).values()
tst_acc = nn.predict(tstX, tstY, acc=True)

print("\nTest Accuracy: ", tst_acc, "\n\n")

fig = plt.figure(figsize=(20, 10), dpi=160, facecolor='w', edgecolor='k')
```

```
fig.suptitle("LSTM\nLearning Rate {} | Momentum = {} | Batch Size  = {} |
 Hidden Layers = {}\n"
             "Train Accuracy: {:.1f} | Validation Accuracy: {:.1f} | Test
              Accuracy: {:.1f}\n "
             .format(eta, alpha, batch_size, size[2:-1],
              train_acc_list[-1], val_acc_list[-1], tst_acc), fontsize=13)

plt.subplot(2, 2, 1)
plt.plot(train_loss_list, "C2", label="Train Cross Entropy Loss")
plt.title("Train Cross Entropy Loss")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.subplot(2, 2, 2)
plt.plot(val_loss_list, "C3", label="Validation Cross Entropy Loss")
plt.title("Validation Cross Entropy Loss")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.subplot(2, 2, 3)
plt.plot(train_acc_list, "C2", label="Train Accuracy")
plt.title("Train Accuracy")
plt.xlabel("Epoch")
plt.ylabel("Accuracy")
plt.subplot(2, 2, 4)
plt.plot(val_acc_list, "C3", label="Validation Accuracy")
plt.title("Validation Accuracy")
plt.xlabel("Epoch")
plt.ylabel("Accuracy")

plt.savefig("q3b.png", bbox_inches='tight')

train_confusion = nn.predict(trX, trY, acc=True, confusion=True)
test_confusion = nn.predict(tstX, tstY, acc=True, confusion=True)

plt.figure(figsize=(20, 10), dpi=160)

plt.subplot(1, 2, 1)
sn.heatmap(train_confusion, annot=True, annot_kws={"size": 8},
 xticklabels=names, yticklabels=names, cmap=sn.cm.rocket_r, fmt='g')
plt.title("Train Confusion Matrix")
plt.ylabel("Actual")
plt.xlabel("Prediction")
plt.subplot(1, 2, 2)
sn.heatmap(test_confusion, annot=True, annot_kws={"size": 8},
 xticklabels=names, yticklabels=names, cmap=sn.cm.rocket_r, fmt='g')
plt.title("Test Confusion Matrix")
plt.ylabel("Actual")
plt.xlabel("Prediction")
plt.savefig("q3b_confusion.png", bbox_inches='tight')

##############################
```

```
alpha = 0.85
eta = 0.01
epoch = 10
batch_size = 32
size = [trX.shape[2], 128, 32, 16, 6]

print("\nGRU Layer\n")

nn = Q3NET(size, 3)
train_loss_list, val_loss_list, train_acc_list, val_acc_list =
 nn.train(trX, trY, eta, alpha, batch_size, epoch).values()
tst_acc = nn.predict(tstX, tstY, acc=True)

print("\nTest Accuracy: ", tst_acc, "\n\n")

fig = plt.figure(figsize=(20, 10), dpi=160, facecolor='w', edgecolor='k')
fig.suptitle("GRU\nLearning Rate {} | Momentum = {} | Batch Size  = {} |
 Hidden Layers = {}\n"
             "Train Accuracy: {:.1f} | Validation Accuracy: {:.1f} | Test
              Accuracy: {:.1f}\n "
             .format(eta, alpha, batch_size, size[2:-1],
              train_acc_list[-1], val_acc_list[-1], tst_acc), fontsize=13)

plt.subplot(2, 2, 1)
plt.plot(train_loss_list, "C2", label="Train Cross Entropy Loss")
plt.title("Train Cross Entropy Loss")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.subplot(2, 2, 2)
plt.plot(val_loss_list, "C3", label="Validation Cross Entropy Loss")
plt.title("Validation Cross Entropy Loss")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.subplot(2, 2, 3)
plt.plot(train_acc_list, "C2", label="Train Accuracy")
plt.title("Train Accuracy")
plt.xlabel("Epoch")
plt.ylabel("Accuracy")
plt.subplot(2, 2, 4)
plt.plot(val_acc_list, "C3", label="Validation Accuracy")
plt.title("Validation Accuracy")
plt.xlabel("Epoch")
plt.ylabel("Accuracy")

plt.savefig("q3c.png", bbox_inches='tight')

train_confusion = nn.predict(trX, trY, acc=True, confusion=True)
test_confusion = nn.predict(tstX, tstY, acc=True, confusion=True)

plt.figure(figsize=(20, 10), dpi=160)
```

```
    plt.subplot(1, 2, 1)
    sn.heatmap(train_confusion, annot=True, annot_kws={"size": 8},
     xticklabels=names, yticklabels=names, cmap=sn.cm.rocket_r, fmt='g')
    plt.title("Train Confusion Matrix")
    plt.ylabel("Actual")
    plt.xlabel("Prediction")
    plt.subplot(1, 2, 2)
    sn.heatmap(test_confusion, annot=True, annot_kws={"size": 8},
     xticklabels=names, yticklabels=names, cmap=sn.cm.rocket_r, fmt='g')
    plt.title("Test Confusion Matrix")
    plt.ylabel("Actual")
    plt.xlabel("Prediction")
    plt.savefig("q3c_confusion.png", bbox_inches='tight')



########################################
#  RUN TEMPLATE
########################################


def ege_ozan_ozyedek_21703374_hw3(question):
    if question == '1':
        q1()
    elif question == '3':
        q3()


question = sys.argv[1]

ege_ozan_ozyedek_21703374_hw3(question)
```